Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study

AKOND RAHMAN, Auburn University, USA

SHAZIBUL ISLAM SHAMIM, Auburn University, USA

DIBYENDU BRINTO BOSE, Virginia Tech, USA

RAHUL PANDITA, Github, USA

Context: Kubernetes has emerged as the de-facto tool for automated container orchestration. Business and government organizations are increasingly adopting kubernetes for automated software deployments. Kubernetes is being used to provision applications in a wide range of domains, such as time series forecasting, edge computing, and high performance computing. Due to such a pervasive presence, Kubernetes-related security misconfigurations can cause large-scale security breaches. Thus, a systematic analysis of security misconfigurations in Kubernetes manifests, i.e., configuration files used for Kubernetes, can help practitioners secure their Kubernetes clusters.

Objective: *The goal of this paper is to help practitioners secure their Kubernetes clusters by identifying security misconfigurations that occur in Kubernetes manifests.*

Methodology: We conduct an empirical study with 2,039 Kubernetes manifests mined from 92 open-source software repositories to systematically characterize security misconfigurations in Kubernetes manifests. We also construct a static analysis tool called Security Linter for Kubernetes Manifests (SLI-KUBE) to quantify the frequency of the identified security misconfigurations.

Results: In all, we identify 11 categories of security misconfigurations, such as absent resource limit, absent `securityContext`, and activation of `hostIPC`. Specifically, we identify 1,051 security misconfigurations in 2,039 manifests. We also observe the identified security misconfigurations affect entities that perform mesh-related load balancing, as well as provision pods and stateful applications. Furthermore, practitioners agreed to fix 60% of 10 misconfigurations reported by us.

Conclusion: Our empirical study shows Kubernetes manifests to include security misconfigurations, which necessitates security-focused code reviews and application of static analysis when Kubernetes manifests are developed.

CCS Concepts: • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: configuration, container orchestration, devops, devsecops, empirical study, kubernetes, misconfiguration, security

1 INTRODUCTION

Container technologies, such as Docker and LXC are gaining popularity amongst information technology (IT) organizations for deploying software applications. For example, PayPal uses 200,000 containers to manage 700 software applications [60]. For managing these containers at scale, practitioners often use automated container orchestration, i.e, the practice of pragmatically managing the lifecycle of containers with tools, such as Kubernetes [59].

Authors' addresses: Akond Rahman, Auburn University, Auburn, AL, USA, akond@auburn.edu; Shazibul Islam Shamim, Auburn University, Auburn, AL, USA, mzs0283@auburn.edu; Dibyendu Brinto Bose, Virginia Tech, Blacksburg, VA, USA, brintodibyendu@vt.edu; Rahul Pandita, Github, Denver, CO, USA, rahulpandita@github.com.

```
securityContext:
  capabilities:
    drop:
      - ALL
  runAsUser: 101
  allowPrivilegeEscalation: true
  ...
```

privileged security context

a

```
rabbitmq:
  username: user
  ## RabbitMQ application password
  password: pFXfkH5cKA
  ...
  ## Value for the RABBITMQ_LOGS environment variable
  ##
  logs: '-'
```

Hard-coded user name

Hard-coded password

b

Fig. 1. Anecdotal evidence of security misconfigurations in Kubernetes manifests. Figure 1a shows an example of a security misconfiguration related to privilege escalation in a `Kind` manifest [59]. Figure 1b shows an example of a hard-coded username and password in a `Helm` manifest [13].

Since its inception in 2014, Kubernetes has established itself as the *de-facto* tool for automated container orchestration [9, 87]. According to Stackrox survey [97], 91% of the surveyed 500 practitioners use Kubernetes for container orchestration. As of Sep 2020, Kubernetes has a market share of 77% amongst all container orchestration tools [99]. Organizations, such as Adidas, Twitter, IBM, U.S. Department of Defense (DOD), and Spotify are currently using Kubernetes for automated container orchestration. Use of Kubernetes has resulted in benefits, e.g., using Kubernetes the U.S. DoD decreased their release time from 3∼8 months to 1 week [19]. In the case of Adidas, the load time for their e-commerce website was reduced by half, and the release frequency increased from once every 4∼6 weeks to 3∼4 times a day [49].

Kubernetes-based container orchestration, similar to every other configurable software, is susceptible to security misconfigurations. However, due to the pervasive nature of Kubernetes-based container orchestration, such misconfigurations can have severe security implications. According to the 2021 'State of Kubernetes Security Report', 94% of 500 practitioners experienced at least one Kubernetes-related security incident, majority of which can be attributed to security misconfigurations [87]. The survey also states Kubernetes-related misconfigurations to "*pose the greatest security concern*" for Kubernetes-based container orchestration [87]. Anecdotal evidence attests to such perceptions: for example, a Kubernetes-related security misconfiguration resulted in a data breach that affected 106 million users of Capital One, a U.S.-based credit card company [44, 100].

Additionally, we observe anecdotal evidence in open-source software (OSS) repositories that provide clues on what categories of security misconfigurations can occur for Kubernetes. In Figure 1 we present two code snippets related to Kubernetes manifests, and mined from OSS repositories [25, 98]. In Figure 1a we observe a misconfiguration, where `allowPrivilegeEscalation` is enabled with `allowPrivilegeEscalation:true`. Enabling `allowPrivilegeEscalation` allows a child process of a container to gain more privileges than its parent process, which malicious users can leverage to gain unauthorized access to the Kubernetes cluster [55]. In Figure 1b, we observe a hard-coded username and password specified in a Kubernetes manifest.

All of the above-mentioned evidence emphasizes the need of inspecting and mitigating security misconfigurations for Kubernetes manifests, i.e., files used to specify configurations for Kubernetes-based orchestration [59]. However, practitioners often lack knowledge needed to mitigate security misconfigurations [15, 55]. A systematic characterization of security misconfigurations can be helpful to gain an understanding of security misconfigurations that appear for Kubernetes. Such characterization is potentially useful to practitioners who can leverage the identified misconfiguration categories for security-focused code review, and apply automated tools to detect and mitigate security misconfigurations that occur for Kubernetes.

The goal of this paper is to help practitioners secure their Kubernetes clusters by identifying security misconfigurations that occur in Kubernetes manifests.

Accordingly, we answer the following research questions:

- **RQ1: *What categories of security misconfigurations occur in Kubernetes manifests?***
- **RQ2: *How frequently do security misconfigurations occur in Kubernetes manifests?***
- **RQ3: *What categories of Kubernetes objects are affected by security misconfigurations?***
- **RQ4: *How do practitioners perceive the identified security misconfigurations in Kubernetes manifests?***

We conduct an empirical study with 2,039 Kubernetes manifests mined from 92 OSS repositories to quantify the frequency of security misconfigurations in OSS Kubernetes manifests. As part of our empirical study we build a security static analysis tool called Security Linter for Kubernetes Manifests (SLI-KUBE). With a qualitative analysis technique called open coding [89], we categorize Kubernetes objects that are affected by security misconfigurations. Further, we submit 133 bug reports to identify practitioners perceptions for the identified security misconfigurations. An overview of our methodology is presented in Figure 2. Source code and datasets used in the paper is available online [80]. SLI-KUBE is also available online [4].

Contributions: We list our contributions as follows:

- A list of security misconfigurations that occur in OSS Kubernetes manifests;
- An empirical evaluation of how frequently security misconfigurations occur in OSS Kubernetes manifests;
- A list of Kubernetes object categories that are affected by security misconfigurations;
- An evaluation of how practitioners perceive the identified security misconfigurations in Kubernetes manifests; and
- SLI-KUBE: A security static analysis tool to quantify the frequency of identified security misconfigurations.

We organize the rest of the paper as follows: in Section 2 we describe the identified security misconfigurations. In Section 3 we provide the methodology to answer RQ2, RQ3, and RQ4. The answers for RQ2, RQ3, and RQ4 are presented in Section 4. The discussion of our empirical study, related work, and limitations of our paper is respectively, provided in Sections 5, 6, and 7. We conclude our paper in Section 8.

2 CATEGORIES OF SECURITY MISCONFIGURATIONS

In this section we address RQ1: *What categories of security misconfigurations occur in Kubernetes manifests?* We first provide the necessary background on Kubernetes manifests in Section 2.1. Next, we describe the methodology to identify security misconfigurations in Section 2.2. Finally, we describe the identified security misconfigurations in Section 2.3.

2.1 Background

Kubernetes is a tool to programmatically manage containers at scale [59]. A Kubernetes installation is colloquially referred to as a Kubernetes cluster [59]. Kubernetes is installed on a physical or virtual machine called the 'host', which runs the Kubernetes API server [59]. The Kubernetes API

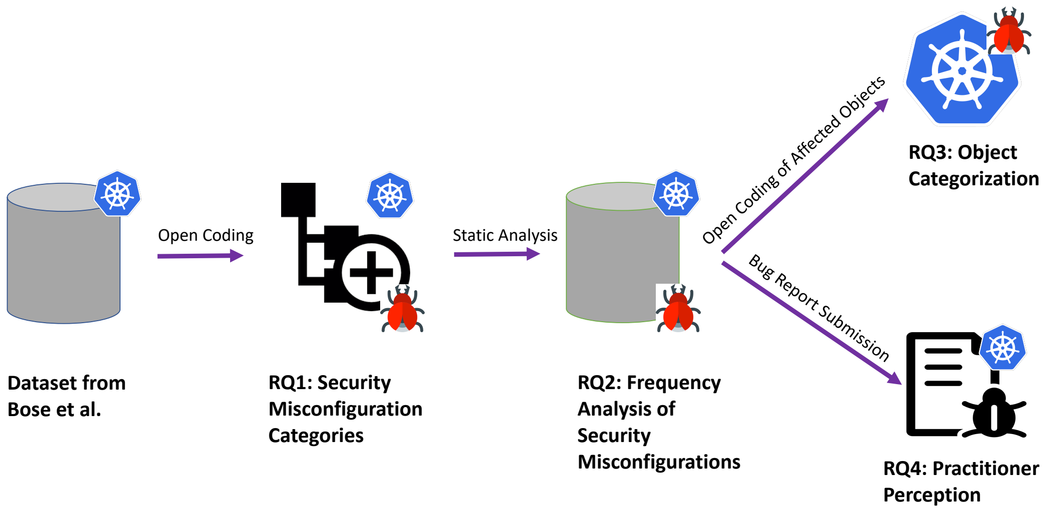

Fig. 2. An overview of our methodology.

server receives and processes HTTP-based API requests. Kubernetes uses a state-based approach where it first queries if the necessary container configurations are consistent with the configurations presented in Kubernetes manifests. If not, with 'kubectl' Kubernetes changes the configurations of containers by applying all the configurations specified in the Kubernetes manifests. A pod is the smallest deployable unit of computing that is created and managed by Kubernetes for container orchestration. A pod logically groups one or multiple containers with shared storage and network resources. For each pod, there exists a specification that applies for all containers grouped by the pod [59]. Specifications for pods can be specified with Kubernetes manifests, i.e., configuration files written in YAML format. For representing the states of orchestrated containers, Kubernetes uses objects. Objects are persistent entities, which allows Kubernetes to know what desired state of orchestration needs to be achieved. Similar to pods, configurations of objects can also be specified with Kubernetes manifests that are written in YAML format. For our empirical study, a Kubernetes manifest can belong to any of the following sub-categories: `Kind` and `Helm`. Throughout the paper a Kubernetes manifest corresponds to either a `Kind` manifest or a `Helm` chart.

`Kind` Manifest: `Kind` manifests are used to specify configurations for objects. `Kind` manifests are executed with Kubernetes-provided utilities, such as `kubectl`. In the case of `Kind` manifests, Kubernetes objects are specified using the `Kind` attribute. `Kind` manifests are different from the `kind` tool [51], which is used to setup and run a Kubernetes locally. Listing 1 shows an example of a pod being specified with a `Kind` manifest. The pod includes one container with the image `hello-world`. We identify `Kind` manifests by inspecting if an YAML manifest includes the following keys: `apiVersion` and `Kind`.

`Helm` Chart: Practitioners can also specify configurations for Kubernetes objects using Helm, a package manager for Kubernetes [13]. Unlike `Kind` manifests, `Helm` charms are executed by the Helm package manager [36]. In `Helm` charts, configuration values can be specified using an YAML manifest called 'values.yaml', which are later used by `templates` [13]. Assignment of a configuration within a `template` confirms that configuration value being used for provisioning [13]. Figure 3 shows an example of using `Helm` manifests to specify configurations. The configuration values for `namespace`

```yaml
kind: Pod
metadata:
  name: example-pod
  labels:
    name: example-pod
    app: example-app
spec:
  containers:
  - image: hello-world
    name: example-pod
```

Listing 1. An example of a Kind manifest.

```
1 # Configuration values specified in a Helm manifest
    ↪ ('values.yaml')
2
3 Dep:
4   namespace: default
5   label: helm-example
6 ...
7
8 # Configuration values used by a Helm template
9 metadata:
10   namespace: {{ .Values.Dep.namespace }}
11   name: {{ .Values.Dep.label }}
12 spec:
13   replicas: {{ .Values.Dep.replicaCount }}
```

Configuration value for `namespace`

Configuration value for `label`

Fig. 3. An example of a Helm chart.

and `label` are later used in a Helm template respectively, in lines#10 (`.Values.Dep.Namespace`) and #11 (`.Values.Dep.replicaCount`). We identify `Helm` charts by inspecting if (i) YAML manifests are labeled as 'values.yaml', and if any of the values are used by YAML manifests are in a directory called 'templates'; or (ii) an YAML manifest resides in the 'template' directory and the 'template' directory contains '.tpl' files.

2.2 Methodology to Identify Security Misconfiguration Categories

We used the qualitative analysis technique - open coding [89] to derive security misconfiguration categories. Open coding is well-suited to identify insights in an under-explored domain, such as Kubernetes security misconfigurations. Furthermore, open coding provides a systematic way to surface similarities across textual artifacts, and group such similarities into categories [89].

As part of the open coding process, *first*, the rater identifies configurations in a Kubernetes manifest. *Second*, the rater inspects the values for each identified configuration to determine if the configuration is in fact a security misconfiguration. While determining misconfigurations, the rater uses the following definition of security misconfiguration provided by the U.S. National Institute of Standards and Technology (NIST) [68] "*A setting within a computer program that violates a configuration policy or that permits unintended behavior that impacts the security posture of a system*". Both raters, who are well-versed on Kubernetes (having used them in practice) initially came up with a list of security misconfigurations that can potentially cause unintended behaviors based on their experience. *Third*, the rater derives categories based on similarities between the identified instances of security misconfigurations. For each identified security misconfiguration category, the rater further checks if the category violates any of the Kubernetes-related security best practices as documented by Shamim et al. [91]. Shamim et al. conducted a grey literature review with 103 Internet artifacts, where they specifically looked into security best practices applicable for Kubernetes. As Internet artifacts are used by industry experts to recommend best practices [82], we assume Shamim et al.'s paper to be used in this content as the paper (i) systematically synthesizes security-related best practices from multiple Internet artifacts, and (ii) is peer-reviewed. Shamim et al.'s paper leveraged a grey literature review with 101 Internet artifacts including multiple artifacts that came out of Snyk [96], where practitioners have discussed the security best practices for Kubernetes. Other artifact sources that were leveraged by Shamim et al. [91] include artifacts authored by practitioners from Google Cloud, Cloud Native Computing Foundation (CNCF), VMWare, Tech Republic, DZone, SonaType, IBM, and Microsoft. We have included the list of Internet artifacts used by Shamim et al. in our replication package [80].

Upon completing the aforementioned three steps, we will derive a list of security misconfiguration categories. In this manner, our identified security misconfigurations convey the message that if identified security misconfigurations are not mitigated, they can permit unintended behaviors.

The first and second authors act as raters, and conduct the open coding process. The first author and second author respectively, has experience in working with Kubernetes for one and two years. Both rater individually manually inspects 1,796 Kubernetes manifests provided by Brinto et al. [9]. Brinto et al. [9]'s dataset includes 1,796 Kubernetes manifests that are modified in 5,193 commits, and collected from 38 OSS repositories. Of the 1,796 Kubernetes manifests, 90% and 10% are respectively, `Kind` and `Helm` manifests. For each Kubernetes manifest, both raters individually apply the aforementioned open coding process.

Upon completion of the open coding process, the first and second authors respectively, identify 11 and 6 categories of security misconfigurations. We compute Krippendorff's α [45] to quantify agreement, similar to prior work in software engineering [6, 29, 86]. The Krippendorff's α is 0.45, indicating *'unacceptable'* agreement [45]. Both raters discussed their disagreements and observed that root cause of their disagreements occur due to the second author missing five categories, identified by the other author. These categories are: activation of `hostIPC`, activation of `hostNetwork`, activation of `hostPID`, capability misuse, and Docker socket mounting. The second rater missed categories because of being unaware of these configurations. Upon discussion, both raters conduct the inspection process again. After completing the inspection process, we calculate Krippendorff's α to be 1.0, indicating *'perfect'* agreement [45]. We use Krippendorff's α instead of Cohen's κ, because Krippendorff's α: (i) emphasizes disagreement leading to more reliability on the achieved agreement rate, and (ii) handles multiple categories [45]. Furthermore, qualitative analysis experts have advocated for the use of Krippendorff's α over Cohen's κ [46, 52].

2.3 Answer to RQ1: Security Misconfiguration Categories

Altogether, we identify 11 categories of security misconfigurations in Kubernetes manifests. An example of each category with a mapping to the violated security practice is presented in Table 1. All the examples presented in Table 1 are obtained from Kind manifests. 'Count' corresponds to the count for the Brinto et al. [9] dataset for each category. Figure 4 presents relative distribution of the identified categories.

Table 1. Examples of Security Misconfiguration Categories

Category (Count)	Violated Practice	Example Code Snippet

Absent Resource Limit (69)	Limit CPU and Memory Quota [91]	`spec:`
		`  containers:`
		`   - name: employee`
		`     image: piomin/employee-service`

Absent `securityContext` (82)	Implementing Pod-specific Policies [91]	`spec:`
		`    containers:`
		`    - name: inventory-container`
		`      image: inventory:1.0-SNAPSHOT`

Activation of `hostIPC` (1)	Implementing Pod-specific Policies [91]	`spec:` `    hostIPC: true`
Activation of `hostNetwork` (13)	Implementing Pod-specific Policies [91]	`spec:` `    hostNetwork: true`
Activation of `hostPID` (2)	Implementing Pod-specific Policies [91]	`spec:` `    hostPID: true`

Capability Misuse (20)	Implementing Pod-specific Policies [91]	`capabilities:`
		`    add:`
		`    - CAP_SYS_ADMIN`
		`    - CAP_SYS_MODULE`

Docker Socket Mounting (4)	Implementing Pod-specific Policies [91]	`- name: dockersocket` `    mountPath: /var/run/docker.sock`
Escalated Privileges for Child Container Processes (1)	Implementing Pod-specific Policies [91]	`allowPrivilegeEscalation: true`
Hard-coded Secret (126)	Authorization & Authentication [91]	`POSTGRES_PASSWORD: VGVzdERCQGhvbWUy`
Insecure HTTP (467)	Enable SSL/TLS Support [91]	`value: http://elastisearch-logging:9200`
Privileged `securityContext` (9)	Implementing Pod-specific Policies [91]	`securityContext:` `    privileged: true`

I. Absent Resource Limit: The category of not specifying resource limits for containers within a pod. A pod is a logical unit that groups a set of containers together for any Kubernetes cluster [50]. With the use of `limits`, the amount of CPU and memory for a pod can be specified. However, if the limits are unspecified, then Kubernetes clusters are susceptible to denial of service attacks [91], as malicious users can increase the flow of traffic, which in turn can lead to unbounded CPU and memory requests [50].

II. Absent `securityContext`: The category of not using `securityContext` while provisioning containers. A lack of `securityContext` is indicative of not applying access control policies for pods, which in turn can provide malicious users the opportunity to gain access into the Kubernetes cluster. Use of `securityContext` is critical to restrict malicious activities that can arise from zero-day vulnerabilities or supply chain attacks for Kubernetes clusters [55].

III. Activation of `hostIPC`: The category of activating `hostIPC` while specifying configurations in Kubernetes manifests. The `hostIPC` configuration controls if containers within a pod can share the inter process communication (IPC) namespace. The IPC namespace provides separation of IPC between the host and containers. If the host's IPC namespace is shared with the container, it would allow processes within the container to see all of IPC communications on the host system. Allowing

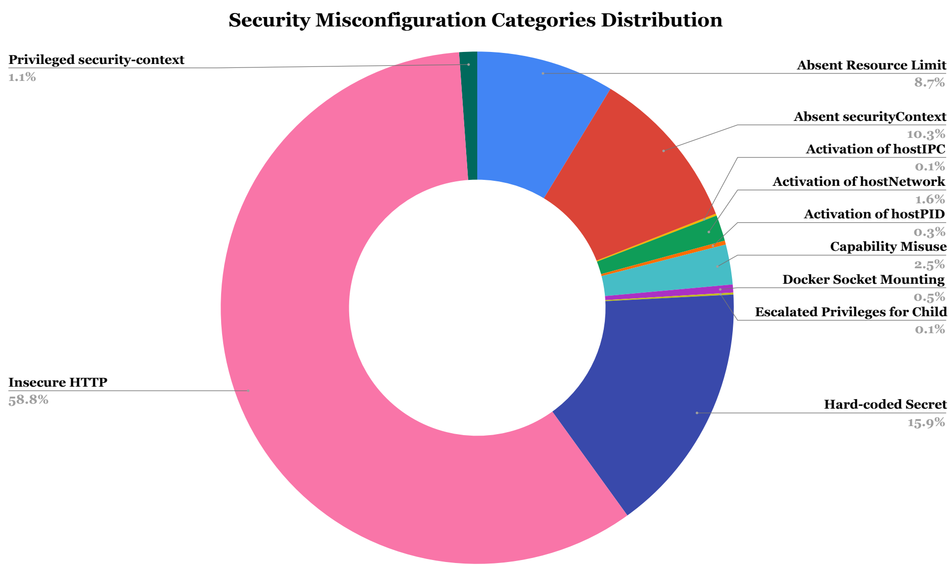

Fig. 4. Distribution of Security Misconfiguration Categories

`hostIPC: true` would not only remove the separation between host and containers, but also allow a malicious user to get access to the host, and observe all processes running on the host [23].

IV. Activation of `hostNetwork`: The category of activating `hostNetwork` while specifying configurations in Kubernetes manifests. For Kubernetes, `hostNetwork` is a configuration that allows a pod to run in the host's network namespace [5]. When a pod is configured with `hostNetwork: true`, the applications running in such a pod can directly see the network interfaces of the host machine where the pod was started. An application that is configured to listen on all network interfaces will in turn be accessible on all network interfaces of the host machine. Use of `hostNetwork: true` allows malicious users to get access to the workloads that are running on the host, and apply packet sniffing tools, such as `tcpdump` [56].

V. Activation of `hostPID`: The category of activating `hostPID` while specifying configurations in Kubernetes manifests. The `hostPID` configuration controls if the containers in a pod can share the host process ID (PID) namespace. The default value is `false`. When `hostPID` is true then a pod has access to the namespace where host process is running. The implication of activated `hostPID` is that it allows a malicious user to find all of the process running on the host, and use that information to conduct malicious activities [18]. The Kubernetes official documentation advises against use of `hostPID: true` stating that if `hostPID: true` is used, in conjunction with process mentoring tools, such as `ptrace` [58], then privilege escalation can occur outside of the container.

VI. Capability Misuse: The category of activating Linux capabilities, which allows malicious users to gain root-level access into a Kubernetes cluster. We observe two categories: (i) misuse with `CAP_SYS_ADMIN`, and misuse with `CAP_SYS_MODULE` configurations.

`CAP_SYS_ADMIN` allows for a wide range of privileged system administration operations, which cannot be performed by a normal user [103]. `CAP_SYS_ADMIN` facilitates container breakouts, i.e., the event where a container user is able to nullify container isolation and access resources, such as system calls on the host machine [55].

With `CAP_SYS_MODULE` capability, Linux kernel modules can be loaded to bypass authorizations in place [55]. Use of `CAP_SYS_MODULE` allows a malicious user to abuse the `SYS_MODULE` capability of Linux to perform container breakout, and retrieve contents of the root Docker host [67].

VII. Docker Socket Mounting: The category of mounting of the Docker socket path by using the `/var/run/docker.sock` configuration. Mounting of Docker socket leaks information about other containers, which can be leveraged by a malicious user. Docker uses a non-networked UNIX socket, and when used in daemon mode, Docker only allows connections from authenticated entities. If this socket is mounted without adequate permissions, then the socket can be used to spin up any container, create new images, or shut down existing containers [12].

VIII. Escalated Privileges for Child Container Processes: The category of allocating privileges for child processes within a container that are higher than that of the parent processes. With `allowPrivilegeEscaltion : true` a child process of a container can gain more privileges than its parent process. The security implication is that malicious users can leverage these child processes to gain unauthorized access to the Kubernetes cluster [11].

IX. Hard-coded Secret: The category of providing hard-coded secrets as configurations in Kubernetes manifests. We identify three sub-categories: (i) hard-coded usernames, (ii) hard-coded passwords, and (iii) hard-coded private tokens. Exposure of hard-coded secrets can be leveraged by malicious users to gain unauthorized access for the Kubernetes cluster. Common Weakness Enumeration (CWE) identifies hard-coded secrets as one of the top 25 security weakness in 2021 [61]. Hard-coded secrets have been attributed to the 2019 Uber data breach [90], the 2020 medical data breach in 2020 [85], and the 2021 D-link breach [69].

X. Insecure HTTP: The category of using HTTP without SSL/TLS certificates to setup URLs or transmit traffic inside and outside the Kubernetes clusters. Without SSL/TLS certificates, the data transmitted across Kubernetes objects are susceptible to man-in-the-middle (MITM) attacks.

XI. Privileged `securityContext`: The category of using privileged `securityContext` in Kubernetes manifests. `securityContext` is used to provide access control configurations for a pod or a container [50]. Examples include but are not limited to: (i) define access control for a Kubernetes object, (ii) apply profiling to restrict capabilities of individual programs running on a Kubernetes cluster, and (iii) allow a certain process to gain more privileges than its parent process. However, due to privileged `securityContext`, all access control features provided by `securityContext` will be obsolete. One Kubernetes expert labeled `privileged: true` as the "*the most dangerous flag in the history of computing*", as this configuration gives the illusion of containerization but in fact disables all security features provided by `securityContext` [55].

We provide a mapping of which security misconfigurations are applicable for `Kind` and `Helm` manifests in Table 2. We observe that majority of the misconfiguration categories are found in `Kind` manifests. All of the 11 identified categories are found in `Kind` manifests, whereas 2 of the 11 categories, namely hard-coded secret and insecure HTTP appear in `Helm` manifests. One possible explanation can be attributed to the dataset we analyzed. Future research can systematically investigate the comparative distribution of security misconfigurations between Helm and Kind manifests.

Table 2. Mapping of Misconfiguration Categories With `Kind` and `Helm` Manifests.

Category	`Kind`	`Helm`
Absent Resource Limit	✓	×
Absent `securityContext`	✓	×
Activation of `hostIPC`	✓	×
Activation of `hostNetwork`	✓	×
Activation of `hostPID`	✓	×
Capability Misuse	✓	×
Docker Socket Mounting	✓	×
Escalated Privileges for Child Container Processes	✓	×
Hard-coded Secret	✓	✓
Insecure HTTP	✓	✓
Privileged `securityContext`	✓	×

Answer to RQ1: We identify 11 categories of security misconfigurations in Kubernetes manifests, which include misconfigurations unique to Kubernetes, such as `absent resource limit`.

3 METHODOLOGY

In this section, we describe the methodology to conduct our empirical study by *first*, describing the construction and evaluation of SLI-KUBE, which we use to quantify the frequency of identified security misconfigurations. *Second*, we provide the methodology to answer RQ2, RQ3, and RQ4.

3.1 Security Linter for Kubernetes Manifests (SLI-KUBE)

We describe the construction and evaluation process of SLI-KUBE respectively, in Sections 3.1.1 and 3.1.2.

3.1.1 SLI-KUBE Methodology. As described in Section 2.1, the flow of configuration data in Kubernetes manifests is unique to Kubernetes itself, which necessitates construction of a static analysis tool that accounts for Kubernetes-specific information flow analysis.

Step-1: Parsing: SLI-KUBE parses Kubernetes manifests into key-value pairs. For each key, a value can be a nested dictionary, or a list, or a single value. In the case of nested dictionaries, SLI-KUBE preserves the hierarchy of the extracted keys for Kubernetes manifest.

Step-2: Rule Matching: From the parsed content of Kubernetes manifests, SLI-KUBE applies rule matching to identify security misconfigurations. The rules needed to identify categories are listed in Table 3. The rules are derived by abstracting code snippets for each misconfiguration category. The rules presented in Table 3 leverage pattern matching similar to prior research [78, 79]. The string patterns used by each rule in Table 3 is provided in Table 4.

Rule Derivation Process: We identify the commonalities in patterns capable of expressing security misconfigurations, and abstract such commonalities as rules to detect misconfigurations. We provide an example in Table 5 to demonstrate our rule derivation process. In the 'Coding Pattern' column, we observe two coding patterns that are instances of over-privileged `securityContext`. In both coding patterns, `privileged` keyword is used to specify the coding pattern. SLI-KUBE can parse both coding patterns as key value pairs, where `privileged` is the key and `true` is the value. In both coding patterns we notice commonality in the key value pairs, which can be abstracted to a rule $isKey(x) \wedge isSecuirtyContext(x) \wedge isPrivileged(x) \wedge isEnabled(x.value)$. We repeat the same abstraction process for other misconfiguration categories.

Step-3: Def-use chain analysis: Static analysis tools are susceptible to generate false positives, if the information flow is disregarded. We mitigate this limitation by applying def-use chain analysis [2], where we track the flow of a misconfiguration within Kubernetes manifests.

SLI-KUBE performs two types of information flow analysis that account for the information flow in `Kind` and `Helm` manifests. In the case of `Kind` manifests, SLI-KUBE recursively applies def-use chain analysis across the nested key-value pairs for each manifest to identify if a security misconfiguration is used by a pod. For `Kind` manifests, SLI-KUBE uses the `spec` tag to identify if a security misconfiguration is used by a pod. In the case of `Helm` manifests, SLI-KUBE applies def-use chain analysis to identify if security misconfigurations that are specified in 'values.yaml' are used by YAML files within the 'templates/' directory.

Table 3. Rules Used by SLI-KUBE

Category	Rule
Absent `securityContext`	$isKey(x) \land isContainer(x) \land \neg isSecuirtyContext(x.value)$
Absent Resource Limit	$(isKey(x) \land (isSpec(x) \lor isContainer(x)) \land \neg(isLimitResources \land (isLimitMemory \land isLimitRequests))))$
Activation of `hostIPC`	$(isKey(x) \land isHostIPC(x) \land isEnabled(x.value)$
Activation of `hostPID`	$(isKey(x) \land isHostPID(x) \land isEnabled(x.value)$
Activation of `hostNetwork`	$(isKey(x) \land isHostNetwork(x) \land (isEnabled(x.value)$
Capability Misuse	$(isKey(x) \land isContainer(x) \land hasCapability(x) \land (isCAPSYSADMIN(x.value) \lor isCAPSYSMODULE(x.value))$
Docker Socket Mounting	$isKey(x) \land isPath(x) \land isDockerSocket(x.value)$
Escalated Privileges for Child Container Processes	$(isKey(x) \land isPrivEscalat(x) \land (isEnabled(x.value)$
Hard-coded Secret	$isKey(x) \land (isUser(x) \lor isPassword(x) \lor isToken(x))$
Insecure HTTP	$isKey(x)(\land isProtocol(x.name) \lor isHTTP(x.value))$
Over-privileged `securityContext`	$isKey(x) \land isSecuirtyContext(x) \land isPrivileged(x) \land isEnabled(x.value)$

3.1.2 Evaluation of SLI-KUBE . Security static analysis tools are subject to empirical evaluation [78, 79]. We use an oracle dataset to evaluate SLI-KUBE's accuracy. A security expert, who is a PhD student, created the oracle dataset. To construct the oracle dataset, we use 240 randomly-selected Kubernetes manifests from the GitLab dataset described in Section 3.2. We use this dataset as it was not used during the open coding process described in Section 2.2. The rater applied closed coding [21] to identify security misconfigurations in a manifest. Closed coding is the process of mapping an entry to a pre-defined category [21]. We do not impose any time limit for the rater to conduct closed coding. We provided the rater a guidebook that included the names, definitions, and examples of each security misconfiguration. The guidebook is available online [80] publicly.

The rater took 50 hours to conduct closed coding. Upon completion of the closed coding process, we apply SLI-KUBE on the 240 Kubernetes manifests collected from 8 repositories. We evaluate SLI-KUBE using precision and recall. Precision refers to the fraction of correctly identified security misconfigurations among the total identified misconfigurations, as determined by SLI-KUBE. Recall refers to the fraction of correctly identified security misconfigurations that have been retrieved by SLI-KUBE. We use Equations 1 and 2 respectively, to calculate precision and recall. In Equations 1 and 2, FN, FP, TN, and TP respectively refers to false negatives, false positives, true negatives, and true positives. For example, if there is 1 instance of absent `securityContext`, and SLI-KUBE identifies that instance without the generating any false positives or false negatives, then SLI-KUBE's recall will be 1.0 according to Equation 2. As another example, if SLI-KUBE identifies that

Table 4. String Patterns Used for Rules in Table 3.

Function	String Pattern
hasCapability()	‘capabilities’
isCAPSYSADMIN()	‘CAP_SYS_ADMIN’
isCAPSYSMODULE()	‘CAP_SYS_MODULE’
isContainer()	‘container’
isDockerSocket()	‘/var/run/docker.sock’
isEnabled()	‘true’
isHostIPC()	‘hostIPC’
isHostNetwork()	‘hostNetwork’
isHostPID()	‘hostPID’
isHTTP()	‘http:’
isLimitMemory()	‘limits’
isLimitRequests()	‘requests’
isLimitResources()	‘resources’
isPath()	‘path’
isPassword()	‘password’
isPrivEscalat()	‘allowPrivilegeEscalation’
isProtocol()	‘protocol’
isPriviledged()	‘privileged’
isSecurityContext()	‘securityContext’
isSpec()	‘spec’
isToken()	‘key’
isUser()	‘user’

Table 5. An Example to Demonstrate the Rule Derivation for ‘Over-privileged `securityContext`’

Coding Pattern	Parsing Output of SLI-KUBE

`-name: neutron-server`
`  securityContext`
`    privileged: true`
	<Key, ‘neutron-server’, <Key, ‘securityContext’, <Key, ‘privileged’, true »>

`-name: cinder`
`  securityContext`
`    privileged: true`
	<Key, ‘cinder’, <Key, ‘securityContext’, <Key, ‘privileged’, true »>

1 instance of absent `securityContext` but generated one false positive then, SLI-KUBE’s precision will be 0.5 according to Equation 1.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

As shown in Table 6, SLI-KUBE’s precision and recall is ≥ 0.90, which gives us the confidence of SLI-KUBE’s ability to detect security misconfigurations automatically, while generating a few false positive instances. We observe SLI-KUBE to generate false positives for hard-coded secrets and insecure HTTP. False positives occur due to pattern matching, e.g., `user_data: ‘cloud-init-parts/generic’` is identified by SLI-KUBE as a hard-coded username, even though a hard-coded username is not being specified. As another example of a false positive is `hostPorts: http:80`, where a port configuration is identified as an instance of insecure HTTP. We do not make any conclusions on the severity of the detected misconfigurations. We acknowledge that SLI-KUBE may perform better with respect to detection, but Snyk may prioritize some misconfigurations better than SLI-KUBE.

Table 6. Evaluation of SLI-KUBE with Oracle Dataset

Category	**Count**	**Precision**	**Recall**
Absent Resource Limit	13	1.0	1.0
Absent `securityContext`	8	1.0	1.0
Activation of `hostIPC`	1	1.0	1.0
Activation of `hostNetwork`	1	1.0	1.0
Activation of `hostPID`	1	1.0	1.0
Capability misuse	20	1.0	1.0
Docker Socket Mounting	1	1.0	1.0
Escalated Privilege for Child Container Processes	1	1.0	1.0
Hard-coded Secret	86	0.82	1.0
Insecure HTTP	214	0.93	1.0
Privileged `securityContext`	8	1.0	1.0
Average	—	0.9	1.0

Table 7. Evaluation of Snyk with Oracle Dataset

Category	**Count**	**Precision**	**Recall**
Absent Resource Limit	13	0.02	1.0
Absent `securityContext`	8	0.0	0.0
Activation of `hostIPC`	1	1.0	1.0
Activation of `hostNetwork`	1	1.0	1.0
Activation of `hostPID`	1	1.0	1.0
Capability misuse	20	1.0	1.0
Docker Socket Mounting	1	1.0	1.0
Escalated Privilege for Child Container Processes	1	1.0	1.0
Hard-coded Secret	86	0.0	0.0
Insecure HTTP	214	0.0	0.0
Privileged `securityContext`	8	1.0	1.0
Average	—	0.64	0.73

Differences Between SLI-KUBE and Existing Tools: We highlight the differences between our tool, SLI-KUBE and existing tools that also analyzes Kubernetes manifests in Table 8. For comparison we select four state-of-the-art security static analysis tools, namely Checkov [10], KubeLinter [48], Datree [24], and Snyk [96]. We inspect the respective documentation for each of them to identify which of the 11 security misconfiguration categories are identified by these tools. Only SLI-KUBE detects all of the 11 categories of security misconfigurations. Checkov, KubeLinter, Datree, and Snyk respectively, is not able to detect 2, 3, 5, and 3 of the 11 security misconfiguration categories. Therefore, the precision and recall will be 0.0 for the categories that Checkov, KubeLinter, Datree, and Snyk are unable to detect. For example as shown in Table 7, we observe Snyk's precision and recall to be 0.0 for absent `securityContext`, insecure HTTP, and hard-coded secrets. The average precision and recall for Snyk is respectively, 0.64 and 0.73.

3.2 Dataset Collection

We quantify the frequency of security misconfigurations by mining OSS projects. We use two data sources: (i) OSS GitLab projects and (ii) OSS GitHub projects. OSS projects hosted on social coding

Table 8. Comparison of SLI-KUBE with Existing Tools

Category	SLI-KUBE	Checkov	KubeLinter	Datree	Snyk
Absent Resource Limit	✓	✓	✓	✓	✓
Absent `securityContext`	✓	✓	×	×	×
Activation of `hostIPC`	✓	✓	✓	✓	✓
Activation of `hostNetwork`	✓	✓	✓	✓	✓
Activation of `hostPID`	✓	✓	✓	✓	✓
Capability Misuse	✓	✓	×	×	✓
Docker Socket Mounting	✓	✓	✓	✓	✓
Escalated Privileges for Child Container Processes	✓	✓	✓	×	✓
Hard-coded Secret	✓	×	✓	×	×
Insecure HTTP	✓	×	×	×	×
Privileged `securityContext`	✓	✓	✓	✓	✓

platforms are susceptible to quality concerns, e.g., users often host projects on GitHub for personal purposes that do not adequately reflect professional software development [63]. To mitigate this issue, in prior work [1, 47, 63, 74], researchers have leveraged a set of attributes of OSS GitHub repositories to identify repositories that are reflective of professional software development. These attributes include but are not limited to count of certain file types [76], count of commits per month [63], and count of contributors [1, 47]. These attributes provide motivation for our criteria to curate OSS repositories:

- **Criterion-1**: At least 10% of the files in the repository must be Kubernetes manifests. By using a cutoff of 10% we seek to collect repositories that contain Kubernetes manifests for analysis. Prior research [41] shows that configuration files can co-locate with source code and test code files. Using this threshold, we assume to identify repositories that have enough Kubernetes configuration files, i.e., manifests for analysis.
- **Criterion-2**: The repository must be available for download.
- **Criterion-3**: The repository is not a clone to avoid duplicates.
- **Criterion-4**: The repository must have ≥ 2 commits per month. Munaiah et al. [63] previously used the threshold of ≥ 2 commits per month to determine which repositories have enough software development activity. We use this threshold to filter repositories with little activity.
- **Criterion-5**: The repository has ≥ 5 contributors. Our assumption is that the criterion of ≥ 5 contributors may help us to filter out irrelevant repositories, such as repositories used for personal use. Prior research [37] has also used the threshold of at least five contributors.
- **Criterion-6**: The repository is not used for a 'toy' project. We consider a project as 'toy' project if description and content of the README file for each projects indicates that the project is used to demonstrate examples, conduct course work, and used as book chapters. Both the first and second author individually conduct this manual inspection. The set of projects that both authors agree to be a toy project is considered as final. By reading the README files of repositories collected with Criterion-5, we also determine if the projects are deployable, i.e., can be downloaded and executed as a software application. Both the first and second author individually conduct this manual inspection. The set of projects that both authors agree to be deployable is considered as final.

Table 9 summarizes how many projects are filtered using our criteria. Attributes of the collected projects are available in Table 10. Altogether we download 92 repositories by cloning the master

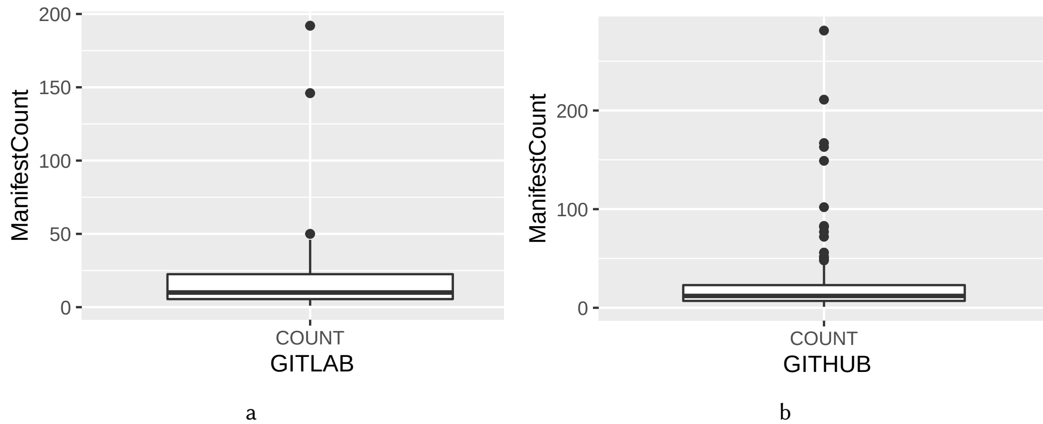

Fig. 5. Distribution of manifests count per repositories respectively for the GitLab and GitHub dataset.

branches on November 2021. We use the GHTorrent dataset hosted on Google Big Query. We run queries on Google Big Query to obtain the initial list of GitHub repositories. In the case of GitLab repositories, we use the GitLab API [33].

For GitHub and GitLab we identify the median count of manifests per repository to respectively, be 10 and 12. The maximum count of manifests per repository is 192 and 281 respectively, for GitLab and GitHub. We provide the full distribution of manifest count per repositories for the GitLab and GitHub datasets in Figure 5.

3.3 RQ2: Frequency of Identified Security Misconfigurations

We answer RQ2 by collecting 2,039 Kubernetes manifests from the 92 repositories. As shown in Table 10, of the 2,039 Kubernetes manifests 449 are obtained from 21 GitLab repositories, and 1,590 Kubernetes manifests from 71 GitHub repositories. Each of the Kubernetes manifest mined from the GitHub and GitLab repositories is either a `Kind` or a `Helm` manifest.

We apply SLI-KUBE on the collected 2,039 Kubernetes manifests to quantify the frequency of identified security misconfigurations. We report four metrics: (i) count, (ii) configuration density, (iii) manifest proportion, and (iv) object proportion. Configuration density corresponds to the

Table 9. Filtering of OSS Projects To Answer RQ_1

	GitHub	**GitLab**
Initial Repo Count	3,405,303	546,000
Criterion-1 ≥10% YAML files)	6,633	8,194
Criterion-2 (Available)	6,512	7,914
Criterion-3 (Non-duplicates)	4,317	5,871
Criterion-4 (Commit/month≥2.0)	1,325	671
Criterion-5 (Contrib. ≥5)	189	44
Criterion-6 (Not Toy Project)	71	21
Final Repo Count	71	21

Table 10. Dataset Attributes

Attribute	GitHub	GitLab
Repositories	71	21
Kubernetes Objects	3,630	827
Kind Manifests	1,508	369
Helm Charts	82	80
Kubernetes Manifests	1,590	449
Contributors	1,187	977
Commits	37,184	15,870
Size (LOC)	148,588	51,512
Duration	9/2015-12/2021 (75 months)	10/2015-12/2021 (74 months)

count of security misconfigurations that appear in every 1,000 lines of code. Manifest proportion corresponds to the proportion of Kubernetes manifests in which at least one instance of security misconfiguration appears. Object proportion corresponds to the proportion of Kubernetes objects that are affected by at least one security misconfiguration. We use these four metrics as each of these metrics can help us contextualize the frequency of security misconfigurations from multiple perspectives. Count provides the occurrences of security misconfigurations. Configuration density measures how many security misconfigurations occur in every 1,000 lines of Kubernetes manifest, and can be used to estimate inspection efforts for Kubernetes manifests. Manifest proportion measure on average how likely a Kubernetes manifest can include at least one instance of security misconfiguration. Object proportion measures on average how many Kubernetes objects are affected by a security misconfiguration. As practitioners seek information on how security issues are used in the code [95], with the metric object proportion, practitioners can assess how many of the Kubernetes objects can be affected by security misconfigurations.

Correlation Between Maturity and Presence of Security Misconfigurations: One possible explanation to the presence of security misconfigurations is maturity, i.e., manifests that are short-lived may tend to include security misconfigurations. We use age to calculate maturity, and use age to evaluate our hypothesis. We calculate age by calculating the difference in days between the first date the manifest was created and the date the manifest was last modified. We hypothesize that the Kubernetes manifests with no security misconfigurations will be more mature, i.e., have longer age than that of Kubernetes manifests with at least one security misconfiguration. Accordingly, we state the following null and alternate hypothesis:

- **Null**: There is no difference in age between Kubernetes manifests with no security misconfigurations and Kubernetes manifests with at least one security misconfiguration.
- **Alternative**: The age of Kubernetes manifests with no security manifests is significantly higher than that of Kubernetes manifests with at least one security misconfiguration.

We reject the null hypothesis if p-value < 0.01 by applying Mann-Whitney U test [53] following Cramer and Howitt's observations [22]. We use Mann-Whitney U test as this test makes no assumptions about the underlying distributions of the data [53].

Correlation Between Development Factors and Presence of Security Misconfigurations: We hypothesize the following metrics related to the development of Kubernetes manifests that correlate with presence of security misconfigurations:

- **IsDeployed**: This metric determines whether or not a manifest is used in a repository, which can be deployed. Our hypothesis is manifests that are part of a deployment-related repository is likely to get more security-focused reviews, and therefore are likely to contain fewer security misconfigurations.

- **Size**: This metric computes the number of lines in a manifest. We hypothesize manifest size to show correlation with presence of security misconfigurations. We take motivation from prior research [81, 104] that demonstrates the co-relation between size and software defects. We hypothesize that the probability of a manifest including a security misconfiguration is higher for manifests that are larger in size.
- **Age**: This metric computes the age of a manifest as measured by the difference between last commit date and first commit date. Prior research [72] has shown age of software artifacts to show correlation with software defects. We hypothesize that the probability of a manifest including a security misconfiguration is higher for manifests that are less mature, i.e., have lower age.
- **Commits**: This metric computes the count of commits made for a manifest. Prior research [64] has shown commits to correlate with the presence of software defects. We hypothesize that the probability of a manifest including a security misconfiguration is higher for manifests that are modified through larger number of commits.
- **Developers**: This metric computes the count of developers who modify a manifest. Prior research [77, 83] has shown developer count to correlate with the presence of software defects. We hypothesize that the probability of a manifest including a security misconfiguration is higher for manifests that are modified by multiple developers than that of fewer developers.
- **Minor contributors**: This metric computes the count of developers who modify $< 5\%$ of the total lines of code for a manifest. Prior research [77, 83] has shown developer count to correlate with the presence of software defects. We hypothesize that the probability of a manifest including a security misconfiguration is higher for manifests that have more minor contributors than others.

We calculate these metrics for all Kubernetes manifests that we obtain from our OSS repositories collected during our filtering criteria.

Quantifying Correlation: We use a logistic regression model [31] to quantify the correlation between presence of security misconfigurations and the aforementioned metrics. In our logistic regression model, the dependent variable is presence of security misconfiguration, with two possible values: '1' indicating presence of a misconfiguration, and '0' indicating absence of a misconfiguration. The independent variables are: '*IsDeployed*', '*size*', '*age*', '*commits*', '*developers*', and '*minor contributors*'. Except for deployment status all metrics are numeric. IsDeployed is a factor variable with two possible outcomes: '1', which means the manifest being part of a repository that is deployed, and '0' that means the manifest is not part of a repository that is not deployed.

Prior to applying the logistic regression, we apply the following recommended practices: (i) apply log transformation to reduce heteroscedasticity [20], and (ii) test if multi-colineraity exists between the independent variables using variable influence factor (VIF) [31]. For our model we report (i) McFadden's R2 [102] value that can estimate our model's explainability, (ii) p-values for each independent variable, and (iii) coefficients, sum of square errors, and deviance for each independent variable.

Following Cramer and Howitt's observations [22], we determine a metric to have a correlation if the p-value for that metric is < 0.01.

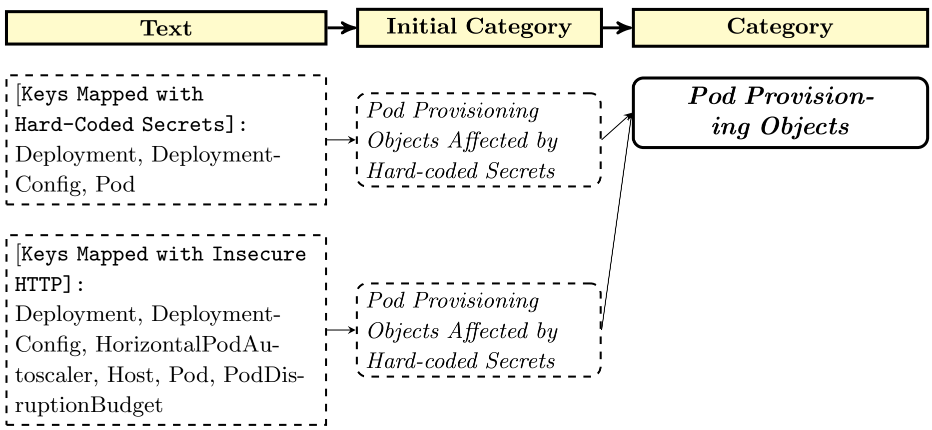

Fig. 6. An example to demonstrate the methodology of applying open coding to determine Kubernetes object categories affected by security misconfigurations.

3.4 RQ3: Kubernetes Objects Affected by Security Misconfigurations

We answer RQ3 using the following steps:

Kind-related Data Separation: *first*, we remove false positive instances generated by SLI-KUBE for both datasets. *Second*, for each of the 11 categories, we extract key values pairs from each Kubernetes manifest. *Third*, we separate key values pairs for the key `Kind`. We identify values for `Kind` because in Kubernetes `Kind` determines the type of Kubernetes object is being provisioned.

Open coding: We identify 70 unique Kubernetes objects respectively, from 256 `Kind` manifests from the above-mentioned step. We apply open coding on the collected 70 Kubernetes object names to determine Kubernetes object categories for which security misconfigurations are specified. Open coding is a qualitative analysis technique to identify categories from structured or unstructured text [89]. For open coding, each rater first reads the definition of each object using the Kubernetes documentation [50]. Next, the rater groups the Kubernetes objects based on definition similarities.

By extracting the values for the `Kind` key in `Kind` manifests, we determine the Kubernetes objects that could be impacted by security misconfigurations. We use Figure 6 to illustrate our process of deriving Kubernetes object categories that are affected by security misconfigurations. Under the 'Text' textbox we observe a set of Kubernetes objects that are affected by a security misconfiguration category. We observe a set of pod-related Kubernetes objects that are affected by two misconfiguration categories: hard-coded secrets and insecure HTTP. As all of these objects are related to provisioning pods, and also affected by security misconfigurations, we group them as one category called 'Pod Provisioning Objects'.

The first and third authors are the two raters, who independently apply open coding as described above. Both raters individually apply open coding for 70 Kubernetes objects. Upon completion of this phase, we record a Krippendorff's α of 0.82, indicating an 'acceptable' agreement [45]. The raters disagree on 2 categories that are resolved using the resolver, i.e., the second author of the paper. The third author identifies two categories not identified by the first author, namely, 'Pod Scaling' and 'Background Process Execution'. The resolver's decision is final on the disagreed upon categories. Our methodology for choosing a resolver is to identify an individual who has worked with Kubernetes in an academic or professional setting. In the department we are unable to find one, and hence used the second author. We use the second author as a resolver as the author is well-versed on Kubernetes, and has used Kubernetes in practice. The second author also has participated in identifying the security misconfiguration categories. Our assumption is that the second author's experience in Kubernetes can help resolve disagreements on what Kubernetes objects are likely to be affected by security misconfigurations.

The resolver read the definition of the two categories and determined if the two categories are stand alone or could be merged with existing categories identified by the first author. The resolver determined that 'Pod Scaling' and 'Background Process Execution' can respectively, be merged with 'Pod' and 'Process Execution', as they both fit the definition of these two categories.

3.5 RQ4: Practitioner Perceptions of Identified Security Misconfigurations

We answer RQ4 using two steps: submit bug reports, and conduct semi-structured interviews where we collect feedback from practitioners directly about SLI-KUBE. We describe these two steps as follows:

3.5.1 Bug Report Submission. We submitted bug reports to gather feedback from practitioners. From the identified misconfigurations with SLI-KUBE, we randomly-selected 242 misconfigurations mined from 43 repositories. Altogether we submit 133 bug reports for which of these 242 misconfigurations. In each bug report, we identify the locations of security misconfigurations, description of the misconfigurations, and possible consequences of the misconfigurations. We ask in the bug report if the practitioner would fix the misconfigurations, or have changed the code to fix the misconfigurations. All of these bug reports are submitted on May 2022. We provide an example of a bug report in Figure 7. The links for all bug reports are available online [80]. Table 11 shows the count of bug reports submitted for each category of security misconfiguration. All bug reports are submitted by May 10, 2022.

3.5.2 Semi-structured Interviews. We conduct semi-structured interviews to get feedback from practitioners. We use randomly-selected 250 email addresses and sent emails to all 250 emails. For our semi-structured interview we used emails from the repositories that we mined and described in Section 3.3. The second author of the paper sent the emails. Upon response and approval, we invited the participants over Zoom. In all, we found 9 interviewees who agree to participate. All interviewees participated via Zoom.

As part of this semi-structured review, we first state the purpose of the interview, demonstrate SLI-KUBE, and then we ask questions. We describe each of these steps below:

Purpose: The purpose of the interview is to understand if `SLI-KUBE` is useful for practitioners to detect security misconfigurations in Kubernetes manifests.

Demonstration of SLI-KUBE: As described by He et al. [35], we perform the following activities to demonstrate SLI-KUBE for each practitioners:

- *Proposition*: Proposition corresponds to describing the goal of the semi-structured interview, which is to obtain feedback from practitioners about the usefulness of SLI-KUBE.
- *Evidence*: Evidence corresponds to the artifacts that are used for the interview. As part of this activity we describe the construction and usage of SLI-KUBE. We also describe verbally the security weakness categories with examples.
- *Method of demonstration*: As part of this activity we showcase the execution of SLI-KUBE where we describe how SLI-KUBE takes input and the output generates. As part of the demonstration

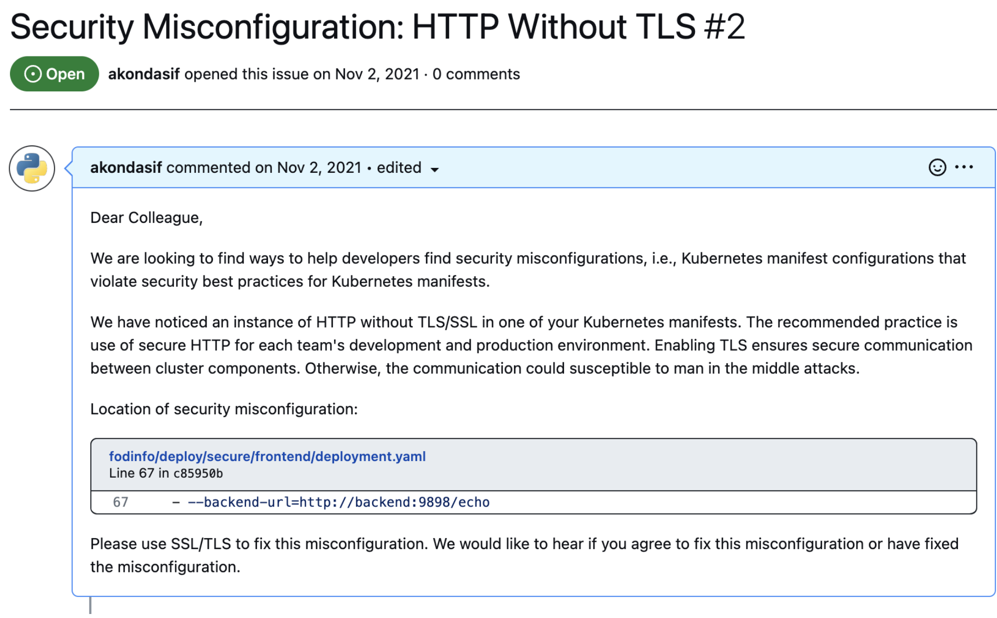

Fig. 7. Example of a bug report submitted used to answer RQ4.

process we ran SLI-KUBE on a repository, showed the output it generates, described the execution flow, and walked through the generated CSV file. While walking through the generated CSV file we discuss the meaning of each column. We showcased the code to demsontrate how SLI-KUBE detects a misconfiguration and applies def-chain analysis.

Questions: Upon demonstration of SLI-KUBE we ask two questions verbatim:

- *Q1-Usefulness*: Do you think SLI-KUBE is useful to detect security misconfigurations in Kubernetes manifests?
- Q2-Transition: How can we transition SLI-KUBE to practice for wide-scale adoption?

We impose no limit on time to answer these questions. We also allowed the participants to talk about any topics that they think is relevant to the answers of the above-mentioned questions.

4 FINDINGS

We provide answers to RQ2, RQ3, and RQ4 respectively, in Sections 4.1, 4.2, and 4.3.

4.1 Answer to RQ2: Frequency of Identified Security Misconfigurations

In this section, we answer **How frequently do security misconfigurations occur in Kubernetes manifests?**

We observe 1,051 instances of security misconfigurations in 2,039 Kubernetes manifests. For the GitLab dataset, at least one security misconfiguration occur in 20.2% of the 449 manifests. For GitHub dataset, 15.7% of the 1,590 manifests include at least one security misconfiguration. A complete breakdown is available in Table 12 where we also report configuration density in the 'Config. Density' column.

Table 11. Count of Submitted Bug Reports for Each Category of Security Misconfigurations

Category	**Count**
Absent Resource Limit	30
Absent `securityContext`	15
Activation of `hostIPC`	1
Activation of `hostNetwork`	7
Activation of `hostPID`	1
Capability misuse	10
Docker Socket Mounting	6
Escalated Privilege for Child Container Processes	2
Hard-coded Secret	29
Insecure HTTP	22
Privileged `securityContext`	10
Total	133

Table 12. Answer to RQ2: Occurrences, Configuration Density, Manifest Proportion, and Object Proportion

	Occurrences		**Config. Density**		**Manifest Prop. (%)**		**Object Prop. (%)**	
Category	**GitLab**	**GitHub**	**GitLab**	**GitHub**	**GitLab**	**GitHub**	**GitLab**	**GitHub**
Absent Resource Limit	10	70	0.25	0.48	2.23	4.4	1.2	2.15
Absent `securityContext`	2	81	0.05	0.5	0.44	4.6	0.2	2.17
Activation of `hostIPC`	0	1	0.0	0.006	0.0	0.06	0.0	0.03
Activation of `hostPID`	0	5	0.0	0.03	0.0	0.3	0.0	3.69
Activation of `hostNetwork`	3	11	0.07	0.07	0.67	0.7	2.29	4.04
Capability Misuse	20	0	0.51	0.0	2.67	0.0	4.35	0.0
Docker Socket Mounting	1	3	0.02	0.02	0.22	0.19	1.2	0.19
Escalated Privileges for Child Container Processes	0	3	0.0	0.02	0.0	0.19	0.0	1.18
Hard-coded Secret	108	111	2.75	0.7	2.22	2.7	0.72	12.4
Insecure HTTP	217	395	5.53	2.7	14.0	8.5	12.9	20.49
Privileged `securityContext`	9	1	0.23	0.006	2.00	0.06	1.8	0.02
Total	370	681	9.43	4.6	20.2	15.7	28.5	46.9

Correlation Between Maturity and Presence of Security Misconfigurations: From our Mann-Whitney U test, we observe p-value = 0.94 and 0.43 respectively, for the GitHub and the GitLab dataset. We cannot reject the null hypothesis, and conclude that maturity of Kubernetes manifests as measured by age is not correlated with presence of security misconfigurations.

Correlation Between Development Factors and Presence of Security Misconfigurations: We present the results of our logistic regression models in Tables 13 and Table 14 respectively, for GitHub and GitLab. In both tables we report the co-efficient estimates, standard errors, p-values, and deviance. For both datasets we observe size to be correlated with presence of security misconfigurations.

For GitHub and GitLab McFadden R2 is respectively, 7.9×10^{-02} and 0.27. This indicates that even though the model for GitLab is well-fitted, the model for GitHub does not fit well. A McFadden R2 value between 0.2 and 0.4 is a good indication of well-fitted model [102]. We also observe a VIF of < 5 for all independent variables for both datasets indicating insignificant multi-colinearity to exist between the independent variables.

Based on our logistic regression analysis for both datasets we conclude size, as measured by lines of code, to correlate with presence of security misconfigurations in Kubernetes manifests. According to our logistic regression analysis for both datasets, the likelihood for including a security misconfiguration is higher for Kubernetes manifests that are larger in size.

Table 13. Logistic Regression Results for GitHub Dataset

Metric	Coeff. Estimate	Error	p-value	Deviance
(Intercept)	-4.9	0.37		
IsDeployed	0.01	0.10	0.90	0.01
Size	0.60	0.05	$< 2 \times 10^{-16}$	203.6
Age	-0.02	0.04	0.59	0.21
Commits	-0.12	0.15	0.41	0.03
Developers	0.21	0.50	0.67	12.9
Minor Contributors	1.37	0.57	0.017	5.84

Table 14. Logistic Regression Results for GitLab Dataset

Metric	Coeff. Estimate	Error	p-value	Deviance
(Intercept)	-5.86	1.07		
IsDeployed	-0.03	0.19	0.87	0.001
Size	1.33	0.12	$< 2 \times 10^{-16}$	228.8
Age	-0.24	0.05	0.02	16.1
Commits	0.34	0.26	0.19	0.71
Developers	-1.26	1.53	0.41	1.37
Minor Contributors	0.32	1.75	0.85	0.03

Answer to RQ2: From our empirical study we identify 1,051 instances of security misconfigurations that affect 13.9% of total 4,707 Kubernetes objects. According to our logistic regression analysis, the likelihood for including a security misconfiguration is higher for Kubernetes manifests that are larger in size.

4.2 Answer to RQ3: Kubernetes Objects Affected by Security Misconfigurations

We identify 6 categories of Kubernetes objects available in `Kind` manifests that are affected by security misconfigurations. A mapping between the identified object category and the security misconfiguration category is provided in Table 15. We describe each of these categories as follows:

Load Balancing for Meshes: The category includes objects that are used to perform load balancing a mesh of services. With Kubernetes, practitioners can implement meshes, i.e., a collection of services to be added and managed with observability in place. In our dataset we observe practitioners using the `Gateway` object, which is provided by Istio to implement service meshes [40]. In the case of service meshes to ensure service reliability, load balancers distribute traffic across multiple services [88]. As shown in Listing 2, the `Gateway` object used for load balancing could be susceptible to MITM attacks if insecure HTTP is used for traffic routing.

Pod Provisioning: This category includes objects that are used to create, scale, manage, and delete all pods within a Kubernetes cluster. A pod is a set of one or multiple containers that share the same storage, same network resources, and specification on how to run these containers [38]. While Kubernetes provides a rich collection of features to manage containers at scale, without the detection and mitigation of security misconfigurations, these containers could provide malicious

```yaml
kind: Gateway
metadata:
  name: cinema-gateway
  namespace: default
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
```

Listing 2. An example of how insecure HTTP is used to provision an Isito `Gateway` object.

actors opportunities to conduct security attacks. In the example presented in Listing 3, all containers that belong to `nfs-server`, will have a privileged `securityContext`, which may lead to container breakouts [55].

```yaml
kind: Pod
...
spec:
  containers:
  - name: nfs-server
    image: call518/oaas-nfs-server:1.0
    securityContext:
      privileged: true
```


Listing 3. Privileged `securityContext` is used to provision a pod using the `Pod` object.

Process Execution: This category includes objects that are used to execute a group of foreground or background process within one or multiple pods. This category include two sub-categories: (i) `DaemonSet` objects, i.e., Kubernetes objects that ensure required background processes are running on all nodes without user intervention [38]; (ii) `CronJob` objects that are used to create Kubernetes jobs on a repeated schedule. In Kubernetes, a job corresponds to the process that executes and re-executes pods until a specified number of pods successfully terminate [38, 50].

In Listing 4 we provide an example, where we observe a `DaemonSet` to be provisioned with activation of `hostNetwork`. As described in Section 2.3, when a pod is configured with `hostNetwork: true`, the applications running in such a pod can directly see the network interfaces of the host machine, which in turn provides malicious users unauthorized visibility. Listing 5 shows how a cron process with the `CronJob` object is used to curl content from an insecure HTTP connection.

Secret: This category includes objects that are used to store and manage secrets, such as usernames, passwords, and private SSH keys. The purpose of the `Secret` object is to efficiently manage secrets that are needed for authorization and authentication without introducing duplicates. In order to secure secrets, Kubernetes stores `Secrets`-related data in a `tmpfs`, which are never written to physical storage [38]. However, hard-coding secrets while provisioning `Secret` objects weakens the security features provided by Kubernetes, as by default "`Secrets` *are stored as unencrypted*

```yaml
kind: DaemonSet
...
  spec:
    serviceAccountName: filebeat
    terminationGracePeriodSeconds: 30
    hostNetwork: true
```

Listing 4. `hostNetwork` is used to provision a process for a pod with the `DaemonSet` object.

```yaml
kind: CronJob
spec:
 containers:
 - name: cronjob
   image: spotify/alpine:latest
   imagePullPolicy: Always
   command:
   - curl
     args:
      - http://bootstorage-svc:5000/api/
      bootstorage/deletelru
```

Listing 5. Insecure HTTP is used to provision a cron process for a pod with the `CronJob` object.

base64-encoded strings and can be retrieved by anyone with API access” [70]. Listing 6 shows a hard-coded username and password to provision a `Secret` object.

```yaml
kind: Secret
metadata:
  name: mongodb-secret
type: Opaque
data:
  username: dXNlcm5hbWU=
  password: cGFzc3dvcmQ=
```

Listing 6. Hard-coded username and password provided for the `Secret` object.

Stateful Applications: This category includes objects that are used to provision stateful applications with `StatefulSet`. Characteristics of stateful applications include but are not limited to: (i) requiring unique network identifiers, (ii) requiring persistent and stable storage, and (iii) updating pods in an ordered and automated rolling manner [50]. Listing 7 shows use of privileged `securityContext` and capability misuse to provision a set of stateful applications.

Traffic Routing: This category includes objects that are used to route service traffic in and out of the Kubernetes cluster. Kubernetes provides a variety of objects that can be used to control how service traffic will be routed within the Kubernetes cluster and outside of the cluster. Examples of such objects include: `Ingress`, `Egress`, `DestinationRule`, and `VirtualService`. While setting up the rules we observe practitioners to use insecure HTTP, which can expose all the traffic generated

```yaml
kind: StatefulSet
...
spec:
  containers:
    - name: cinder
      image: call518/oaas-newton
      ...
      securityContext:
        privileged: true
        capabilities:
          add:
            - CAP_SYS_ADMIN
```

Listing 7. `CAP_SYS_ADMIN` and over-privileged `securityContext` is used to provision a stateful application with the `StatefulSet` object.

and managed by their Kubernetes clusters to be susceptible to MITM attacks. We provide an example in Listing 8.

```yaml
kind: DestinationRule
...
spec:
  host: istio-policy.istio-system.svc.cluster.local
  trafficPolicy:
    connectionPool:
      http:
```


Listing 8. Insecure HTTP is used to provision routing of network traffic with the `DestinationRule` object.

Table 15. Mapping of Kubernetes Object Categories and Security Misconfiguration Categories

Kubernetes Object	**Misconfiguration**
Load Balancing for Meshes	Insecure HTTP
Secret	Hard-coded secret
Stateful Applications	Privileged `securityContext`, Activation of `hostNetwork`, Capability Misuse
Pod	Absent `securityContext`, Absent Resource Limit, Activation of `hostPID`, Activation of `hostIPC`, Activation of `hostNetwork`, Escalated Privileges for Child Container Processes, Insecure HTTP
Process Execution	Activation of `hostPID`, Activation of `hostIPC`, Activation of `hostNetwork`, Docker Socket Mounting
Traffic Routing	Insecure HTTP

Answer to RQ3: Six categories of Kubernetes objects are affected by security misconfigurations: load balancing for meshes, secret, stateful applications, pods, process execution, and traffic routing.

4.3 Answer to RQ4: What are the practitioner perceptions of the identified security misconfigurations?

We answer RQ4 by describing the responses obtained from bug reports (Section 4.3.1) and semi-structured interviews (Section 4.3.2).

4.3.1 Answer to RQ4 - Bug Reports. In this section, we answer **How do practitioners perceive the identified security misconfigurations in Kubernetes manifests?** As of September 15, 2022, we obtain 10 responses to our bug reports for 242 instances with a response rate of 4.1%. Out of 10, we observe practitioners agree with the reported 6 misconfiguration instances. The most agreed upon category are: activation of `hostIPC`, activation of `hostPID`, and Docker socket mounting. The least agreed upon category is insecure HTTP. A complete breakdown of reported practitioner perception is provided in Figure 8. We have not obtained any responses for the following categories: absent resource limit, absent `securityContext`, activation of `hostNetwork`, capability misuse, escalated privileges for child container processes, and privileged `securityContext`.

In the case of insecure HTTP category, practitioners stated the following reasons on why they disagreed. For one instance of insecure HTTP one practitioner mentioned that the identified insecure HTTP instance is invalid as it is used internally "*Thanks, but this is an internal call so I'm not too worried.*". Another practitioner disagreed with an instance of insecure HTTP as the practitioner assumed that the developed manifest will used with cert-manager, and thus the reported instance is invalid: "*TLS is fully supported in podinfo [the manifest name] when using a service mesh*". Another practitioner discarded an instance of insecure HTTP assuming the submitted bug report was generated by a bot: "*I know you are a bot*". In the case of a hard-coded secret, one practitioner mentioned that these are default values stating "*default values in k8s files*". The above-mentioned statements from disagreeing practitioners also suggest lack of awareness, e.g., if another practitioner comes across a manifest with a hard-coded secret, then that practitioner can perceive hard-coded secrets to be acceptable [79].

Nuanced Perspectives of Insecure HTTP: Figure 8 shows practitioners to disagree mostly with insecure HTTP. One possible explanation is that inherently traffic within pods can be protected with TLS support. For example, Istio internally uses TLS for inter-service communication [88], and therefore detected instances of insecure HTTP that are managed with Istio will not be perceived positively by practitioners. Despite reported disagreements we advocate for the mitigation of insecure HTTP instances as both local and remote sites that use HTTP can be insecure [7].

Our response rate is low, which can be attributed to a lack of monetary incentives [76, 94], practitioners' negative biases for static analysis alerts [42, 78, 79] as well as for submitted bug reports related to security static analysis alerts [79]. Survey response rate in cybersecurity and software engineering research can respectively, be as low as 3% [66] and 6% [94]. We mitigate the limitation of low response rate for bug reports by conducting semi-structured interviews that we have discussed in the next section.

4.3.2 Answer to RQ4 - Interviews. From our semi-structured interview we observe all 9 practitioners to find SLI-KUBE useful for detecting security misconfigurations in Kubernetes manifests. In Table 16 we report their responses along with their reported experience in working with Kubernetes. As shown in the 'Usefulness of SLI-KUBE' column in Table 16, all practitioners agreed that security misconfiguration categories detected by SLI-KUBE are valid, and useful to secure the Kubernetes-based installations. For example, I8 said "*Some DevOps folks don't care about security so tools like this [SLI-KUBE] are helpful as these tools can automatically find security issues*". I8 further added "*Before this [SLI-KUBE] I didn't know we should scan insecure http. Now I understand the importance*

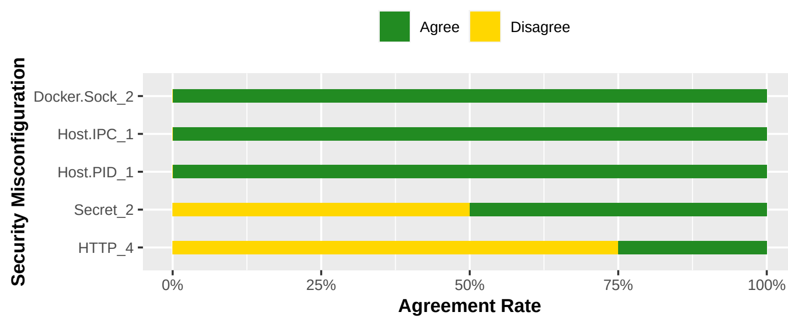

Fig. 8. Answer to RQ4: Practitioner perception of identified security misconfigurations

of checking insecure HTTP". Similar positive enthusiasm was expressed by I7 who said "*In general it [SLI-KUBE] looks super cool. Good work. I am a Kubernetes developer myself and I see the beauty of it*".

From the response of the second question we obtain the following activities to transition SLI-KUBE to practice:

- **CI pipeline integration**: Multiple interviewees (I3, I6, I7, I8, I9) suggested SLI-KUBE's seamless integration in a continuous integration (CI) pipeline as one possible improvement. Currently SLI-KUBE runs as a Python application. With integration in a CI-based pipeline, such as in Jenkins, a practitioner may find SLI-KUBE more useful. According to I8, "*I will be happy if this tool can be easily integrated into CI systems, such as Jenkins*". I7 stated "*For what I have seen people do not run static analysis tools on their own local machines because people are lazy. The way it is 100% sure that people will use is in the CI*". I3 further added, "*We can easily integrate this tool in QA and staging. If you have something like the scripts then we can add the tool to our CI pipeline*".
- **Kubernetes integration**: Our interviewees (I1, I2, I5) also suggested the integration of SLI-KUBE inside Kubernetes itself. I1 observed that SLI-KUBE will be better utilized if the tool is already available as part of Kubernetes. Whenever a manifest is executed Kubernetes will check if any of the 11 security misconfiguration categories appear. I1's views were echoed by I2 who suggested two other alternatives on how to integrate SLI-KUBE into Kubernetes. One option is admissions controller that uses the Kubernetes API. An admission controller is a program that intercepts requests to the Kubernetes API server before making a Kubernetes object persistent [59]. I5 discussed how containers can be leveraged for Kubernetes-based integration: "*What you could also do if you would be able to put the checking tool [SLI-KUBE] in a container and access the tool via Kubernetes API. Many etcd tools work like that. So there are ways to enhance it [SLI-KUBE] without changing the tools too much*".
- **Severity-based prioritization**: Interviewees (I4, I5, I6, I8) recommended severity-based prioritization for SLI-KUBE so that it not only reports the occurrences of security misconfigurations but also prioritizes these occurrences based on severity. I8 stated "*All the categories are important, but if the users can understand the priorities then that would be good*". Interviewees also provided hints on what are the highly severe misconfiguration categories that deserve prioritization. For example, according to I5 the highly severe misconfiguration categories are: capability misuse, activation of `hostNetwork`, activation of `host IPC`, and Docker socket mounting. I6 identified the

Table 16. Interviewee Profile

ID	Experience (Years)	Job Title	Duration (Minutes)	Usefulness
I1	2	Consultant	28.2	YES
I2	7	SRE	33.0	YES
I3	3	SRE	30.3	YES
I4	5	SRE	38.5	YES
I5	4	Developer	30.2	YES
I6	2	Developer	36.4	YES
I7	2	Developer	26.3	YES
I8	3	SRE	26.4	YES
I9	1	Developer	9.1	YES

following as highly severe misconfiguration categories: escalated privileges for child container processes, privileged `securityContext`, and hard-coded secret.

- **Flexibility for users**: Currently, to use SLI-KUBE a user needs to provide a directory of Kubernetes manifests, which is later analyzed to identify occurrences of all 11 categories security misconfigurations. This could be limiting as pointed out by multiple interviewees (I2, I4, I9). I2 stated "*Different companies want different things, allowing their people to run their own checks. Having the flexibility to control what checks to run is beneficial*". I2 hinted at the use of policy languages, such as Cue [1] and Rego [2]. I4 suggested SLI-KUBE to also consider Kubernetes objects that have already been provisioned: "*I believe it would be useful if it [SLI-KUBE] would also work on existing Kubernetes objects. A cluster's configuration can be different from the configurations of Kubernetes manifests. I don't believe this would be an issue as you can get YAML files from running Kubernetes installations. I guess you would have to connect from the script with the cluster and then you may have to use 'kubeconfig' from the local environment*".

> *Answer to RQ4: From our semi-structured interviews, we observe all interviewed practitioners to find SLI-KUBE to be useful in identifying security misconfigurations. For our submitted bug report, we observe a 60% agreement for the 10 security misconfigurations for which we obtained responses.*

5 DISCUSSION

We discuss the implications of our findings in this section:

5.1 Kubernetes Objects Affected by Security Misconfigurations

According to Martin and Hausenblas [55], in order to facilitate 'vanilla' deployments for a wide range of users, "*Kubernetes has been designed to be historically with minimum security features*". Hence, practitioners should be aware of the security misconfigurations, and how these misconfigurations can be detected and mitigated while developing Kubernetes manifests. Yet, our empirical study shows that practitioners include security misconfigurations, >= 15.7% of the manifests include at least one security misconfiguration. These misconfigurations impact the Kubernetes objects that are pivotal to provision Kubernetes clusters, such as objects used in load balancing, and objects used for stateful applications. Given the fact that Kubernetes is being used to provision applications in a wide

[1] https://cuelang.org/docs/integrations/k8s/
[2] https://www.kubermatic.com/blog/opa-rego-in-a-nutshell/

range of domains, such as forecasting [93], smart grids [65], edge computing [92] [101], electronic vehicles [43], machine learning [17], and high performance computing [54, 62], unmitigated security misconfigurations leave these provisioned software application open to attacks from malicious users. As organizations rely on Kubernetes to automate their software supply chain [14], unmitigated security weaknesses in Kubernetes manifests can lead to security attacks against Kubernetes-based software supply chain.

5.2 Implications for Practitioners

Our findings show that security misconfigurations in Kubernetes manifests are quite prevalent. For the GitHub and GitLab dataset the proportion of Kubernetes manifests is respectively 15.7% and 20.2%. We recommend the following practices in this regard:

5.2.1 Application of Security Static Analysis. We recommend practitioners to use static analysis to identify security misconfigurations in Kubernetes manifests. Practitioners can use our tool SLI-KUBE to identify the 11 categories of security misconfigurations. We also advocate the scanning to be conducted before being pushed to a repository, otherwise, misconfigurations related to objects, such as `Secrets` will be uploaded to the repositories.

5.2.2 Mitigation Strategies. While static analysis can detect security misconfigurations, further efforts are needed to mitigate the detect instances. To that end, we suggest the following recommendations to remove security misconfigurations:

Absent `securityContext`: With the `securityContext` configurations, adequate access control should be applied for all containers that are managed by a pod.

Absent Resource Limit: With configurations, such as `cpu`, `memory`, `request`, and `limit`, all provisioned containers' CPU and memory should be bounded.

Activation of `hostIPC`: Practitioners should use a `PodSecurityPolicy` that ensures `hostIPC` is set to `false` for all pods.

Activation of `hostPID`: Practitioners should use a `PodSecurityPolicy` that ensures `hostPID` is set to `false` for all pods.

Activation of `hostNetwork`: Instead of using `hostNetwork: true` for gaining access to the host network, practitioners can use `docker run –user` from the Kubernetes console [3].

Capability Misuse : Instead of using `CAP_SYS_ADMIN` and `CAP_SYS_MODULE` with no restrictions, practitioners should apply the principle of least privilege to allow certain containers with limited Linux capabilities. Practitioners are encouraged to leverage configurations, such as `–cap-drop` and `–cap-add` to limit which containers have what capabilities.

Docker Socket Mounting: Practitioners should avoid the exposure of Docker daemon socket via `/var/run/docker.sock`. In the case, the use of the Docker daemon socket is necessary, the socket should be used in read-only fashion in a secured manner using either a HTTPS-based encrypted socket or a secure web proxy [26, 71].

Escalated Privileges for Child Container Processes: To configure pods, `allowPrivilegeEscalation` should always be set to `false`.

Hard-coded Secret: Practitioners should use secret management tools, such as Hashicorp Vault [34] and Bitnami Sealed Secrets [8, 84] with the recommended secret management practices [75].

Insecure HTTP: For traffic routing TLS/SSL should always be enabled for HTTP. Kubernetes provides the `certificates.k8s.io API`, which allows for TLS support where TLS certificates are managed by a Certificate Authority [50].

Privileged `securityContext` : The `privileged` configuration for `securityContext` should remain `false`. In the case a container needs certain capabilities, practitioners can use Kubernetes capabilities [3].

5.2.3 Affected Kubernetes Objects. Objects are pivotal for orchestrating containers with Kubernetes. Objects specified in manifests tell Kubernetes what is the desired state that the orchestrated containers need to be. By characterizing the objects that are affected by security misconfigurations we gain an understanding what types of computing infrastructure are being impacted. For example, from Table 15 we observe security misconfigurations to affect pods that are used to manage containers. Answer to RQ3 show that critical computing infrastructure are impacted security misconfigurations, and thus needs to be mitigated with secure development of Kubernetes manifests. Table 15 shows that security misconfigurations detected by SLI-KUBE are used by Kubernetes objects used to manage critical container-based infrastructure, which could be helpful for practitioners to be more aware of security misconfigurations in Kubernetes manifests.

5.3 Implications for Researchers

We describe the implications for researchers in the following subsections:

5.3.1 Opportunities for Future Research. One contribution of our empirical study is the development of SLI-KUBE, which could be of interest to researchers for future work. With SLI-KUBE, researchers can investigate if combinations of the security misconfigurations can lead to novel attacks. Researchers can investigate to what extent existing vulnerability repair techniques can be applied to repair Kubernetes-related security misconfigurations, and what other novel techniques need to be proposed and evaluated. Results presented in Tables 13 and 14 provide an opportunity for researchers to understand and characterize the presence of security misconfigurations by considering socio-technical factors unique to Kubernetes development.

5.3.2 Benchmark-related Implications. Any emerging domain benefits from empirical benchmarks to facilitate further research and transition research to practice [28]. Our empirical findings stated in Section 4 will directly contribute in establishing empirical benchmarks for Kubernetes security. In particular, our paper is the first to show the frequency of security misconfigurations in Kubernetes manifests through systematic mining of software repositories. Future research can investigate to what extent the frequency of identified security misconfigurations are generalizable for other datasets obtained from proprietary domains. Furthermore, SLI-KUBE can also be used as part of developing empirical benchmarks that can further advance the science of Kubernetes-based container orchestration.

5.3.3 Transition to Practice. While SLI-KUBE has shown promise in detecting security misconfigurations, further research and development efforts need to be pursued to transition SLI-KUBE from a research tool to a practitioner tool, which can be easily integrated into mainstream IDEs, such as Visual Studio Code. Accomplishing the following activities might be of interest to researchers and practitioners for transitioning SLI-KUBE to practice:

[3]https://jamesdefabia.github.io/docs/user-guide/containers/

- Expand the derived taxonomy presented in Section 2.3 by including more security misconfiguration categories and more container orchestration tools. Replicating our methodology presented in Section 2.2 could be a starting point to accomplish this activity.
- Reduce false positives through generation of novel techniques. Empirical evidence presented in Section 3.1 shows that SLI-KUBE as well as Snyk are prone to generating false positives. Therefore, further research is needed to reduce false positives in detecting security misconfigurations.
- Generate repairs of security misconfigurations so that detected misconfigurations are mitigated effectively.
- Transition SLI-KUBE for practitioner use by executing recommendations listed in Section 4.3.2:
  - Integrate SLI-KUBE to CI pipelines;
  - Integrate SLI-KUBE to Kubernetes internally;
  - Prioritize misconfiguration categories reported by SLI-KUBE; and
  - Provide flexibility for SLI-KUBE users.

6 RELATED WORK

Our paper is closely related to existing research in Kubernetes, which remains an under-explored area. Casalicchio et al. [16] analyzed 97 academic publications, and concluded security area to be an under-explored research domain for Kubernetes. To address this gap, researchers have conducted empirical studies: e.g., Shamim et al. [91] conducted a grey literature review using a qualitative analysis of 103 Internet artifacts and derived 11 security best practices for configuring and managing Kubernetes cluster. As another example, Bose et al. [9] identified the presence of security defect-related commits in Kubernetes OSS repositories. While these studies are a good starting point, we observe a lack of research related to empirical studies in the area of Kubernetes security misconfigurations.

Our paper is also closely related with empirical studies focused on secure software development, which is becoming commonplace [27, 30, 32, 39, 57, 78, 79, 105]. Meng et al. [57] have investigated the prevalence of insecure coding practices in Java by observing accepted answers in StackOverflow. Islam et al. [39] identified coding anti-patterns with security implications for enterprise Java applications. Ghafari et al.[32], Gadient et al. [30], and Rahkema et al. [73] in separate studies quantified the presence of vulnerable code in software ecosystems, such as Android [30, 32] and Swift [73]. In the domain of infrastructure as code (IaC), Rahman et al. [78] applied static analysis to quantify frequency of security weaknesses in Ansible [79], Chef [79], and Puppet scripts [78]. However, techniques that apply for IaC scripts, such as Ansible, Chef, and Puppet scripts are not applicable for Kubernetes manifests, as the syntax and semantics of Kubernetes manifests is different to that of IaC scripts [59].

The aforementioned discussions demonstrates a lack of empirical research in the area of Kubernetes security misconfigurations, which we address in our paper.

7 THREATS TO VALIDITY

We discuss the limitations of our paper as follows:

Conclusion Validity: Our derivation of security misconfiguration categories used in Section 2 are limited to the dataset provided by Bose et al. [9]. We mitigate this limitation by allocating raters with experience in Kubernetes who inspect each of the 1,796 Kubernetes manifests. Furthermore, we characterize the 92 repositories used for our empirical study as 'open-source software', which may give the impression that these are well-curated software projects whose software is open-source. However, our repositories might not be reflective of such well-curated projects. We mitigate this limitation by adding another criterion, where the first author manually inspects any available README files and descriptions of obtained GitHub and GitLab projects.

Our criterion to determine the deployment-ability can include repositories that are used for other experimental goals, such as staging [82] and not necessarily deploying an application. We mitigate this limitation through manual inspection of the README files of each repository.

SLI-KUBE may generate false negatives and false positives when applied on other datasets. Such limitation can bias the results presented for RQ2 in Section 4.1. We mitigate this limitation by evaluating SLI-KUBE using an oracle dataset as discussed in Section 3.1.2.

The Kubernetes objects reported in Section 4.2 are limited to the datasets mined from GitHub and GitLab. If the same methodology is applied for other datasets collected from proprietary domains, additional categories of Kubernetes objects could be obtained, which are not reported in Section 4.2. Furthermore, we have not differentiated between objects that are native to Kubernetes and that come from third party controllers.

Construct Validity: SLI-KUBE is a static analysis tool that applies def-use chain analysis to identify a security misconfiguration. SLI-KUBE does not leverage mesh-related semantics, and as a result may detect instances of insecure HTTP that are irrelevant. Furthermore, our use of `Kind` manifests also include Istio manifests that are used for service meshes, which might yield objects unique to Istio, and impact the results of RQ3. We mitigate this limitation by generating categories of affected objects with open coding. We acknowledge that our criteria are not comprehensive and can miss the additional curation that we could have obtained using the criterion of a license file.

Our bug report response is low, which can be limiting to conclude the usefulness of SLI-KUBE for practitioners. We mitigate this limitation by conducting semi-structured interviews with 9 practitioners. All practitioners agreed that SLI-KUBE is useful to detect security misconfigurations in Kubernetes manifests.

External Validity: Our datasets are constructed by mining OSS projects. Our findings may not generalize for proprietary datasets. Furthermore, our empirical study is susceptible to the limitation that we cannot claim the repositories used are reflective of production Kubernetes-based deployments, and therefore, our findings may not generalize to Kubernetes manifests used for production in IT organizations.

Internal Validity: While constructing the oracle dataset the rater may have expectations on the outcomes that could potentially impact the closed coding process. We mitigate the limitation by using a rater who is not an author of this paper.

8 CONCLUSION

Kubernetes has become the go-to tool to implement the practice of automated container orchestration. While Kubernetes has yielded benefits for IT organizations, security misconfigurations can make Kubernetes-based software deployments susceptible to security attacks. To aid practitioners in securing their Kubernetes clusters we have conducted an empirical study with 2,039 Kubernetes manifests. We identify 11 categories of security misconfigurations for Kubernetes manifests, which can be used to conduct security-focused code review for Kubernetes manifests. Using SLI-KUBE we identify 1,051 instances of security misconfigurations in 2,039 Kubernetes manifests. We observe 6 categories of Kubernetes objects affected by security misconfigurations, which include Kubernetes objects used to provision pods and traffic routing. We also observe that practitioners agree with 60% of 10 reported instances of security misconfigurations.

Based on our findings, we recommend the application of security-focused code review and static analysis to identify security misconfigurations, so that unmitigated misconfigurations are not leveraged by the malicious users to conduct Kubernetes-related security breaches. Our derived taxonomy—that includes 11 categories of security misconfigurations—can be useful for practitioners to identify configurations that have security implications. Also, with SLI-KUBE, practitioners can also identify where security misconfigurations are located, and what Kubernetes objects are affected. In this manner, practitioners will obtain further context about where a security misconfiguration occurs, and how they are used to orchestrate containers with Kubernetes objects.

Our empirical study also lays the groundwork for further research in the domain of container orchestration, e.g., systematic creation of benchmarks, generation and mitigation of novel attacks, development of automated techniques that can repair security misconfigurations, and transition SLI-KUBE to practice. Results of RQ2 showcases the variation in frequency of security misconfiguration categories, which can further be explored and replicated for proprietary datasets. We hope our empirical study will advance the science of secure container orchestration.

ACKNOWLEDGMENTS

The research was partially funded by the U.S. National Science Foundation (NSF) award # 2247141 and # 2209636. We thank the PASER group at Auburn University for their valuable feedback. Special thanks to Farhat Lamia Barsha for her help on creating the oracle dataset.

REFERENCES

[1] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 2018. We Don'T Need Another Hero?: The Impact of "Heroes" on Software Development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. ACM, New York, NY, USA, 245–253. https://doi.org/10.1145/3183519.3183549

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[3] Akihiro Suda. 2020. [CVE-2020–15257] Don't use –net=host . Don't use spec.hostNetwork . https://medium.com/nttlabs/dont-use-host-network-namespace-f548aeeef575. [Online; accessed 09-Jan-2022].

[4] akondrahman. 2022. akondrahman/sli-kube. https://hub.docker.com/repository/docker/akondrahman/sli-kube

[5] Ales Nosek. 2017. Accessing Kubernetes Pods from Outside of the Cluster. https://alesnosek.com/blog/2017/02/14/accessing-kubernetes-pods-from-outside-of-the-cluster/. [Online; accessed 18-Jan-2022].

[6] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering* 22, 6 (2017), 3057–3087.

[7] Kayce Basques. 2015. Why HTTPS matters. https://web.dev/why-https-matters/. [Online; accessed 12-Jan-2022].

[8] bitnami labs. 2022. bitnami-labs/sealed-secrets. https://github.com/bitnami-labs/sealed-secrets. [Online; accessed 10-Jan-2022].

[9] Dibyendu Brinto Bose, Akond Rahman, and Shazibul Islam Shamim. 2021. 'Under-reported' Security Defects in Kubernetes Manifests. In *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 9–12.

[10] bridgecrew. 2022. checkov. https://www.checkov.io/4.Integrations/Kubernetes.html. [Online; accessed 12-May-2022].

[11] bridgecrew. 2022. Ensure containers do not run with AllowPrivilegeEscalation. https://docs.bridgecrew.io/docs/ensure-containers-do-not-run-with-allowprivilegeescalation. [Online; accessed 10-Jan-2022].

[12] bridgecrew. 2022. Limit mounting Docker socket daemon in a container. https://docs.bridgecrew.io/docs/bc_k8s_26. [Online; accessed 20-Jan-2022].

[13] Steve Buchanan, Janaka Rangama, and Ned Bellavance. 2020. Helm Charts for Azure Kubernetes Service. In *Introducing Azure Kubernetes Service*. Springer, 151–189.

[14] Jim Bugwadia. 2022. A MAP for Kubernetes Supply Chain Security. https://nirmata.com/2022/03/15/a-map-for-kubernetes-supply-chain-security/. [Online; accessed 02-Nov-2022].

[15] Canonical. 2021. Kubernetes and cloud native operations report 2021. https://juju.is/cloud-native-kubernetes-usage-report-2021

[16] Emiliano Casalicchio and Stefano Iannucci. 2020. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience* 32, 17 (2020), e5668.

[17] Swati Choudhary et al. 2021. Kubernetes-Based Architecture For An On-premises Machine Learning Platform. (2021).

[18] Chris Pisano. 2019. Limiting Pod Privileges: hostPID. https://medium.com/@chrispisano/limiting-pod-privileges-hostpid-57ce07b05896. [Online; accessed 21-Jan-2022].

[19] CNCF. 2020. With Kubernetes, the U.S. Department of Defense Is Enabling DevSecOps on F-16s and Battleships. https://www.cncf.io/case-study/dod/

[20] Patricia Cohen, Stephen G West, and Leona S Aiken. 2014. *Applied multiple regression/correlation analysis for the behavioral sciences*. Psychology press.

[21] Benjamin F Crabtree and William L Miller. 1999. *Doing qualitative research*. sage publications.

[22] Duncan Cramer and Dennis Laurence Howitt. 2004. *The Sage dictionary of statistics: a practical resource for students in the social sciences*. Sage.

[23] DataDogHQ. 2022. Host's IPC namespace is not shared. https://docs.datadoghq.com/security_platform/default_rules/cis-docker-1.2.0-5.16/. [Online; accessed 19-Jan-2022].

[24] datree. 2022. datree. https://hub.datree.io/built-in-rules#containers. [Online; accessed 14-May-2022].

[25] dghubble. 2022. dghubble/go-twitter. https://github.com/dghubble/go-twitter. [Online; accessed 12-Jan-2022].

[26] Docker. 2022. Daemon socket option. https://docs.docker.com/engine/reference/commandline/dockerd/. [Online; accessed 19-Jan-2022].

[27] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[28] Norman Fenton and James Bieman. 2014. *Software metrics: a rigorous and practical approach*. CRC press.

[29] Erin Friess. 2019. Scrum Language Use in a Software Engineering Firm: An Exploratory Study. *IEEE Transactions on Professional Communication* 62, 2 (2019), 130–147. https://doi.org/10.1109/TPC.2019.2911461

[30] Pascal Gadient, Mohammad Ghafari, Patrick Frischknecht, and Oscar Nierstrasz. 2019. Security code smells in Android ICC. *Empirical software engineering* 24, 5 (2019), 3046–3076.

[31] Andrew Gelman and Jennifer Hill. 2006. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press.

[32] Mohammad Ghafari, Pascal Gadient, and Oscar Nierstrasz. 2017. Security smells in android. In *2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 121–130.

[33] GitLab. 2022. API Documentation. https://docs.gitlab.com/ee/api/. [Online; accessed 01-June-2022].

[34] Hashicorp. 2022. Manage Secrets & Protect Sensitive Data. https://www.vaultproject.io/. [Online; accessed 19-Jan-2022].

[35] Wei He, Jianyang Ding, Xiaomei Shen, Xinyu Han, and Longli Tang. 2021. A Survey on Software Reliability Demonstration. *IOP Conference Series: Materials Science and Engineering* 1043, 3 (jan 2021), 032008. https://doi.org/10.1088/1757-899x/1043/3/032008

[36] Helm. 2021. The package manager for Kubernetes. https://helm.sh/. [Online; accessed 03-Nov-2021].

[37] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[38] Bilgin Ibryam and Roland Huß. 2019. *Kubernetes patterns: reusable elements for designing cloud-native applications*. O'Reilly Media.

[39] Mazharul Islam, Sazzadur Rahaman, Na Meng, Behnaz Hassanshahi, Padmanabhan Krishnan, and Danfeng Daphne Yao. 2020. Coding practices and recommendations of spring security for enterprise applications. In *2020 IEEE Secure Development (SecDev)*. IEEE, 49–57.

[40] Istio. 2022. Istio/Gateway. https://istio.io/latest/docs/reference/config/. [Online; accessed 10-Jan-2022].

[41] Yujuan Jiang and Bram Adams. 2015. Co-evolution of infrastructure and source code-an empirical study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 45–55.

[42] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877

[43] Abdukodir Khakimov, Aleksandr Loborchuk, Ibodulaev Ibodullokhodzha, Dmitry Poluektov, Ibrahim A. Elgendy, and Ammar Muthanna. 2020. Edge Computing Resource Allocation Orchestration System for Autonomous Vehicles. In *The 4th International Conference on Future Networks and Distributed Systems (ICFNDS)* (St.Petersburg, Russian Federation) *(ICFNDS '20)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. https://doi.org/10.1145/3440749.3442594

[44] Derek Kortepeter. 2019. U.S. lawmakers eye AWS role in Capital One data breach. https://techgenix.com/aws-capital-one-data-breach/

[45] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.

[46] Klaus Krippendorff and Joseph L Fleiss. 1978. Reliability of binary attribute data.

[47] Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Tim Menzies. 2018. What is the Connection Between Issues, Bugs, and Enhancements?: Lessons Learned from 800+ Software Projects. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. ACM, New York, NY, USA, 306–315. https://doi.org/10.1145/3183519.3183548

[48] kubelinter. 2022. kubelinter. https://docs.kubelinter.io/#/generated/checks. [Online; accessed 13-May-2022].

[49] Kubernetes 2020. Kubernetes User Case Studies. https://kubernetes.io/case-studies/

[50] Kubernetes. 2021. Production-Grade Container Orchestration. https://kubernetes.io/

[51] kubernetes sigs. 2022. kind. https://kind.sigs.k8s.io/. [Online; accessed 02-June-2022].

[52] Matthew Lombard, Jennifer Snyder-Duch, and Cheryl Campanella Bracken. 2010. Practical resources for assessing and reporting intercoder reliability in content analysis research projects. (2010).

[53] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. http://www.jstor.org/stable/2236101

[54] G Manoj Kumar, Rohit Danti, Odso Amit, R Guru Raghavendra, BR Kiran, and HA Sanjay. 2022. Performance Evaluation of HPC Application in Containerized and Virtualized Environment. In *Emerging Research in Computing, Information, Communication and Applications*. Springer, 793–803.

[55] Andrew Martin and Michael Hausenblas. 2021. *Hacking Kubernetes: Threat-Driven Analysis and Defense*. O'Reilly Media.

[56] max. 2020. host ports and hostnetwork: the NATty gritty. https://lambda.mu/hostports_and_hostnetwork/. [Online; accessed 17-Jan-2022].

[57] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure coding practices in java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. 372–383.

[58] Michael Kerrisk. 2021. ptrace(2) — Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html. [Online; accessed 21-Jan-2022].

[59] S. Miles. 2020. *Kubernetes: A Step-By-Step Guide For Beginners To Build, Manage, Develop, and Intelligently Deploy Applications By Using Kubernetes (2020 Edition)*. Independently Published. https://books.google.com/books?id=M4VmzQEACAAJ

[60] Mirantis. 2021. What are the primary reasons your organization is using Kubernetes? https://www.mirantis.com/cloud-case-studies/paypal/

[61] MITRE. 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html. [Online; accessed 21-Jan-2022].

[62] Subrota Kumar Mondal, Rui Pan, HM Kabir, Tan Tian, and Hong-Ning Dai. 2021. Kubernetes in IT administration and serverless computing: An empirical study and research challenges. *The Journal of Supercomputing* (2021), 1–51.

[63] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* (2017), 1–35. https://doi.org/10.1007/s10664-017-9512-6

[64] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 284–292. https://doi.org/10.1109/ICSE.2005.1553571

[65] Amal Nammouchi, Phil Aupke, Andreas Kassler, Andreas Theocharis, Viviana Raffa, and Marco Di Felice. 2021. Integration of AI, IoT and Edge-Computing for Smart Microgrid Energy Management. In *2021 IEEE International Conference on Environment and Electrical Engineering and 2021 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I CPS Europe)*. 1–6. https://doi.org/10.1109/EEEIC/ICPSEurope51590.2021.9584756

[66] Jarmo Nevala. 2018. Cybersecurity situation analysis-Survey in Central Finland 2016-2018. (2018).

[67] Nishant Sharma. 2020. Docker Container Breakout: Abusing SYS_MODULE capability! https://blog.pentesteracademy.com/abusing-sys-module-capability-to-perform-docker-container-breakout-cf5c29956edd. [Online; accessed 14-Jan-2022].

[68] NIST. 2021. misconfiguration. https://csrc.nist.gov/glossary/term/misconfiguration

[69] NJCCIC Advisory. 2021. D-Link Issues Patch for Hard-Coded Password Router Vulnerabilities. https://www.cyber.nj.gov/alerts-advisories/d-link-issues-patch-for-hard-coded-password-router-vulnerabilities. [Online; accessed

12-Jan-2022].
[70] NSA. 2021. Kubernetes Hardening Guidance. https://media.defense.gov/2021/Aug/03/2002820425/-1/-1/1/CTR_KUBERNETESHARDENINGGUIDANCE.PDF. [Online; accessed 10-Jan-2022].
[71] OWASP. 2022. Docker Security Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/. [Online; accessed 11-Jan-2022].
[72] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and software technology* 55, 8 (2013), 1397–1418.
[73] Kristiina Rahkema and Dietmar Pfahl. 2020. Empirical study on code smells in iOS applications. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. 61–65.
[74] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. 2018. Characterizing the Influence of Continuous Integration: Empirical Results from 250+ Open Source and Proprietary Projects. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics* (Lake Buena Vista, FL, USA) *(SWAN 2018)*. ACM, New York, NY, USA, 8–14. https://doi.org/10.1145/3278142.3278149
[75] Akond Rahman, Farhat Lamia Barsha, and Patrick Morrison. 2021. Shhh!: 12 Practices for Secret Management in Infrastructure as Code. In *2021 IEEE Secure Development Conference (SecDev)*. 56–62. https://doi.org/10.1109/SecDev51306.2021.00024
[76] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3377811.3380409 pre-print: https://akondrahman.github.io/papers/icse20_acid.pdf.
[77] Akond Rahman, Effat Farhana, and Laurie Williams. 2020. The 'as code'activities: development anti-patterns for infrastructure as code. *Empirical Software Engineering* 25, 5 (2020), 3430–3467.
[78] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 164–175.
[79] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2021), 1–31.
[80] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2022. Verifiability Package for Paper. https://figshare.com/s/bced7c8353853a983cd7. [Online; accessed 20-August-2022].
[81] Akond Rahman and Laurie Williams. 2019. Source Code Properties of Defective Infrastructure as Code Scripts. *Inf. Softw. Technol.* 112, C (aug 2019), 148–163. https://doi.org/10.1016/j.infsof.2019.04.013
[82] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. 2015. Synthesizing Continuous Deployment Practices Used in Software Development. In *Proceedings of the 2015 Agile Conference (AGILE '15)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/Agile.2015.12
[83] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. 432–441. https://doi.org/10.1109/ICSE.2013.6606589
[84] Kasun Rajapakse. 2021. Sealed Secrets with Kubernetes. https://enlear.academy/sealed-secrets-with-kubernetes-a3f4d13dbc17. [Online; accessed 10-Jan-2022].
[85] Fahmida Rashid. 2020. Medical Data Leaks Linked to Hardcoded Credentials in Code. https://www.beyondtrust.com/blog/entry/hardcoded-and-embedded-credentials-are-an-it-security-hazard-heres-what-you-need-to-know. [Online; accessed 02-Jan-2022].
[86] Päivi Raulamo-Jurvanen, Simo Hosio, and Mika V. Mäntylä. 2019. Practitioner Evaluations on Software Testing Tools. In *Proceedings of the Evaluation and Assessment on Software Engineering* (Copenhagen, Denmark) *(EASE '19)*. Association for Computing Machinery, New York, NY, USA, 57–66. https://doi.org/10.1145/3319008.3319018
[87] RedHat. 2021. State of Kubernetes Security Report. https://www.redhat.com/en/resources/state-kubernetes-security-report
[88] Sachin Manpathak. 2019. Kubernetes Service Mesh: A Comparison of Istio, Linkerd, and Consul. https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/. [Online; accessed 20-Jan-2022].
[89] Johnny Saldana. 2015. *The Coding Manual for Qualitative Researchers*. SAGE.
[90] Julian Schwarz. 2019. Hardcoded and Embedded Credentials are an IT Security Hazard – Here's What You Need to Know. https://www.beyondtrust.com/blog/entry/hardcoded-and-embedded-credentials-are-an-it-security-hazard-heres-what-you-need-to-know. [Online; accessed 02-July-2021].
[91] M. Islam Shamim, F. Ahamed Bhuiyan, and A. Rahman. 2020. XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. In *2020 IEEE Secure Development (SecDev)*. IEEE Computer Society, Los Alamitos, CA, USA, 58–64. https://doi.org/10.1109/SecDev45635.2020.00025
[92] Behshid Shayesteh, Chunyan Fu, Amin Ebrahimzadeh, and Roch Glitho. 2021. Auto-adaptive Fault Prediction System for Edge Cloud Environments in the Presence of Concept Drift. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 217–223. https://doi.org/10.1109/IC2E52221.2021.00037

[93] Greg M Silverman, Himanshu S Sahoo, Nicholas E Ingraham, Monica Lupei, Michael A Puskarich, Michael Usher, James Dries, Raymond L Finzel, Eric Murray, John Sartori, et al. 2021. Nlp methods for extraction of symptoms from unstructured data for use in prognostic covid-19 analytic models. *Journal of Artificial Intelligence Research* 72 (2021), 429–474.

[94] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 89–92. https://doi.org/10.1109/CHASE.2013.6614738

[95] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 248–259. https://doi.org/10.1145/2786805.2786812

[96] Snyk. 2022. snyk. https://snyk.io/security-rules/kubernetes/. [Online; accessed 15-May-2022].

[97] Stackrox. 2021. Kubernetes and Container Security and Adoption Trends. https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/

[98] stefanprodan. 2022. stefanprodan/podinfo. https://github.com/stefanprodan/podinfo. [Online; accessed 12-Jan-2022].

[99] T4. 2020. Container Platform Market Share, Market Size and Industry Growth Drivers, 2018 - 2023. https://www.t4.ai/industries/container-platform-market-share

[100] Twain Taylor. 2020. 5 Kubernetes security incidents and what we can learn from them. https://techgenix.com/5-kubernetes-security-incidents/

[101] Oana-Mihaela Ungureanu, Călin Vlădeanu, and Robert Kooij. 2021. Collaborative Cloud - Edge: A Declarative API orchestration model for the NextGen 5G Core. In *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 124–133. https://doi.org/10.1109/SOSE52839.2021.00019

[102] Michael R Veall and Klaus F Zimmermann. 1996. Pseudo-R2 measures for some common limited dependent variable models. *Journal of Economic surveys* 10, 3 (1996), 241–259.

[103] Vowneee. 2021. Kubernetes clusters should not grant CAPSYSADMIN security capabilities. https://serverfault.com/questions/1068292/kubernetes-clusters-should-not-grant-capsysadmin-security-capabilities. [Online; accessed 15-Jan-2022].

[104] Hongyu Zhang. 2009. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance*. 274–283. https://doi.org/10.1109/ICSM.2009.5306304

[105] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.